Toshiba Global Commerce Solutions
Programming Guide

TOSHIBA
Leading Innovation

Store Integrator Programming Guide

Version 3 Release 5

G362-0562-05

Toshiba Global Commerce Solutions
Programming Guide

TOSHIBA
Leading Innovation

Store Integrator Programming Guide

Version 3 Release 5

G362-0562-05

Note

Before using this information and the product it supports, be sure to read Safety Information–Read This First, Warranty Information, Uninterruptible Power Supply Information, and the information under Notices.

June 2014

This edition applies to Version 3 Release 5 of the licensed program Toshiba Global Commerce Solutions Store Integrator Programming Guide (program number 5639-P71) and to all subsequent releases and modifications until otherwise indicated in new editions.

If you send information to Toshiba Global Commerce Solutions you grant Toshiba Global Commerce Solutions a nonexclusive right to use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Toshiba Global Commerce Solutions, Inc. 2013

© Copyright IBM Corporation 2005, 2010

Contents

Figures vii

Tables ix

Safety xi

About this guide xiii
Who should read this guide xiii
How to use this guide xiii
Using the hyperlinks xiii
Where to find more information xiv

Notice statements xv

Summary of changes xvii
June 2014 xvii
February 2013 xvii
October 2011 xvii

Chapter 1. Overview 1
AEF architecture overview 2
Real or virtual sessions 2
AEF programming tasks 4
POSProvider/POSServer API overview 5
POSProvider/POSServer architecture overview 5
Point of Sale Business Component overview 6
POSBC control and sales-related functions 6
POSBC logical printer functions 7
POSBC logical receipt functions 8
POSBC architecture overview 8
POS Sales Application 9
AEF API 9
POSBC 9
Known Issues and Limitations 9

Chapter 2. Understanding the architecture. 11
Infrastructure 11
AEFSession 11
AEFSessionFactory 12
AEFSessionPool 13
SessionServer 13
LoadBalancer 14
AEFBase 14
Remote Method Invocation (RMI) heartbeats 15
AEFException 15
Component configurations 15
Configuration properties 17
Automation Provider 19
Error handling modes 20
Setting the error handling mode 21
Identifier arguments 22
Info objects 23
Configuration properties 24
POSAutomationProvider 26
Operator 26
Customer 26
Transaction 26
SalesTransaction 26
LineItem 27
Item 27
Coupon 27
Discount 27
Points 27
Tender 28
Using remote references 28
Automation API 28
Action classes 28
Property conditions 31
Bad conditions 33
ConditionLock 34
ObjectDetectorLock 34
AEF error handling 34
Determining the error key 35
Error helper classes 36
Data Provider 40
Properties 40
Event and listener types 43
Extending POSAppEvents 47
Listener considerations 48
Service Provider 50
POSProvider/POSServer API 51
POSProvider 54
POSProviderFactory 55
KioskPOSServer 55
POSServerFactory 55
POSRequestFactory 55
SalesTransaction 55
POSInfo objects 55
Payment integrity 57
Exceptions and errors 60
Client exception handling 60
Configuration 61
POS Business Component 62
JIFServer 63
Message Processing 64
BusinessComponentListenerImpl 64
XML remote procedure call 64
XML validation 65
BusinessComponent 65
Configurability 65
Interaction diagrams 66
Incoming message/event 66
Outgoing message/event 67
POSBC main classes 68
Virtual receipts 69
Interaction examples 71
POSBC configuration 79
Failover 80

POSBC transaction recovery 87

Chapter 3. Programming with the AEF 93
Getting started 93
Setting up the development environment . . . 93
Setting up Eclipse to develop Store Integrator extensions 94
Using the quick start sample 105
AEF 107
Infrastructure 107
Data Provider 111
Automation Provider 125
Service Provider 137
Programming with the POSBC 146
Detecting that a device is in use 148
Client code written in C connecting with the POSBC 148
GS1 DataBar and the POSBC 156
Coding guidelines 157
Using error logging 157
Writing an error helper 159
Handling exceptions 167
Building, debugging and packaging 175
Using properties file configuration 175
Packaging your code 177
Using the 4690 debugger for POS applications under CSS 178

Chapter 4. AEF enabling a POS application 181
AEF enabling a POS application 181
AEF configuration settings 181
POS application hooks 182
Supermarket Application payment system hooks 183
Enabling user extensions and additional features 186

Chapter 5. Programming with the POSProvider/POSServer API 189
Getting started 189
Setting up the development environment . . . 189
Using the sample file 189
POSProvider/POSServer infrastructure 189
Creating a ProviderRequest 190
Creating a POSProvider 190
Testing the POSProvider connection 191
Performing a sale 191
Performing a payment 191
Completing the transaction 192
Releasing POSProvider 192
Force Release 192

Appendix A. SA user exit code 193

Appendix B. GSA user exit code . . . 195

Appendix C. Error handling 197
Overview 197
Good and bad conditions 197
AEFErrorHandler 197
Using the Configuration Bundle 199
Automation Bundle - error handling 199
The Class Bundle 200
Using the Error Bundle 200
Function Code Bundle 200
Key Sequence Bundle - error handling 201
The State Bundle 201
Substate Bundle 201
Error bundle 201
Error helpers 202

Appendix D. AEF property files 203
Editing property files 203
Automation bundle 203
Class Bundle 203
Classes.properties 203
appclasses.properties 203
Configuration Bundle 204
The Error bundle 204
Using the Function code bundle 204
Internationalization bundle 204
Key sequence bundle 204
Logon bundle 205
Session bundle 205
Session.properties information 205
State bundle 205
Substate bundle - AEF 205
Tender map bundle 205

Appendix E. Data Provider properties 207
CouponProperties 207
CustomerProperties 209
DiscountProperties 209
ItemSalesProperties 210
LineItemProperties 212
OperatorAuthorizationProperties 212
Authorization properties (common) 212
Authorization properties (GSA) 213
Authorization properties (SA and ACE) . . . 214
OptionsProperties 215
PointsProperties 218
POSDeviceProperties 218
StoreOptionsProperties 222
TenderProperties 227
TerminalOptionsProperties 227
TransactionStatusProperties 229
TransactionTotalsProperties 232
WorkstationStatusProperties 233

Appendix F. POS application XML events 235
CashReceipt event 235
cashReceipt 235
Coupon event 236
coupon 236
DataEvent 239
Using dataEvent 239
Discount events 240
transactionDiscount 240
lineItemDiscount 241

ItemSales event 242
itemEvent 242
Operator events 246
signOn 246
signOff 248
lock 249
unlock 249
Customer event 249
Customer 249
Options event 251
Options 251
Points event 259
Points 259
Report event 260
report 260
Scale event 260
scale 260
Tender event 260
Tender 260
Tender declined 261
Tender or tenderDeclined 261
TransactionStatus Events 263
transactionStart 263
transactionVoid 265
transactionSuspended 266
transactionEnd 267
transactionUpdate 269
transactionTaxChange 269
TransactionTotals event 270
transactionTotals 270
WorkstationStatus event 271
workstationStatus 271
TransactionReplayComplete event 271
transactionReplayComplete 271

Appendix G. SALogonActionImpl source example 273

Appendix H. POS device support . . . 287
POS device support 287
POS printer 287

Notices 291
Telecommunication regulatory statement 292
Electronic emission notices 292
Federal Communications Commission statement 292
Industry Canada Class A Emission Compliance statement 292
Avis de conformité à la réglementation d'Industrie Canada 292
European Union Electromagnetic Compatibility (EMC) Directive Conformance Statement . . . 292
European Community (EC) Mark of Conformity Statement 293
Germany Class A statement 293
Australia and New Zealand Class A statement 294
People's Republic of China Class A electronic emission statement 294
Russian Electromagnetic Interference (EMI) Class A statement 294
Japanese Electrical Appliance and Material Safety Law statement 294
Japanese power line harmonics compliance statement 295
Japan Voluntary Control Council for Interference Class A statement 295
Japan Electronics and Information Technology Industries Association (JEITA) statement . . . 295
Korean communications statement 295
Taiwanese Class A compliance statement . . . 295
Cable ferrite requirement 296
Electrostatic discharge (ESD) 296
Product recycling and disposal 296
Battery return program 297
For Taiwan: 297
Taiwan contact information 298
For the European Union: 298
For California: 299
Flat panel displays 299
Monitors and workstations 299
Trademarks 299

Index 301

Figures

1. Example of file directory xiv
2. AEFSession architecture 2
3. 4690 terminal session 3
4. 4690 CSS sessions 4
5. Major system components for POSProvider/POSServer 6
6. POSBC architecture 9
7. AEFSession components 12
8. Retrieving a session using a SessionServer 14
9. Warning example 15
10. Typical in-store configuration 16
11. Execute a transaction sequence 20
12. Automation provider error handling code 22
13. Identifier interface hierarchy 23
14. Info object interfaces 24
15. Condition array example 32
16. Data Provider property hierarchy 42
17. POSDataProvider event hierarchy 44
18. AEFDateTimeListener default values 45
19. GenericEventListener sample code 46
20. Flow of events 50
21. Flow of requests 51
22. Retrieving the POSProvider from the POSProviderFactory 53
23. Using the POSProvider to process a sales transaction 54
24. POSRequest objects flowchart 56
25. POSResult objects flowchart 57
26. Message flow 67
27. Message flow 68
28. XSD sample 69
29. POSReceiptEvents 70
30. Examples of POSReceipt and POSReceiptEvents (Part 1 of 2) 72
31. Examples of POSReceipt and POSReceiptEvents (Part 2 of 2) 73
32. AddReceiptLinesRequest XML data 74
33. Returned POSReceiptEvent 75
34. CashReceiptEvents as a result of AddItem 76
35. CashReceiptEvent as a result of AddItem 77
36. VoidLastItemRequest 77
37. CashReceiptEvent as a result of AddItem request 77
38. CashReceiptEvents as a result of automatic totaling 78
39. CashReceiptEvent as a result of VoidItem 79
40. Diagram of backup controller fails during normal operation. 83
41. Diagram 2 of backup controller fails during normal operation. 84
42. Default timeout value 89
43. Configuring your Java SDK in Eclipse. . . . 95
44. Add JRE 96
45. Creating your Eclipse Project. 97
46. Java Settings 98
47. Java Settings Default output folder. 99
48. Configuring the Build Path 100
49. Setting JDK Compliance 101
50. Compiler compliance level 102
51. Exporting your eclipse project 103
52. Exporting resources 104
53. Exporting to si_user.jar 105
54. Example of SampleGUI client command 106
55. SampleGUI initial progress panel 106
56. SampleGUI main panel 107
57. Displaying a Data Provider property value 112
58. AEFPropertyChangeListener example 114
59. TillPositionEvent 116
60. TillPositionEventImpl Part 1 of 2) 117
61. TillPositionEventImpl Part 2 of 2) 118
62. SampleGUI action classes 126
63. Example Solution View 139
64. Connecting code example (Part 1 of 8) 149
65. Connecting code example (Part 2 of 8) 150
66. Connecting code example (Part 3 of 8) 151
67. Connecting code example (Part 4 of 8) 152
68. Connecting code example (Part 5 of 8) 153
69. Connecting code example (Part 6 of 8) 154
70. Connecting code example (Part 7 of 8) 155
71. Connecting code example (Part 8 of 8) 156
72. Sample error logging program 159
73. Simple Error Helper (Part 1 of 3) 162
74. Simple Error Helper (Part 2 of 3) 163
75. Simple Error Helper (Part 3 of 3) 164
76. AEF-enabled application 183

Tables

1. AEF API categories 11
2. AEFBase configuration properties 17
3. AEFSession configuration properties 18
4. Automation Provider configuration properties 24
5. Base action classes 28
6. ACE action classes 29
7. GSA action classes 29
8. SA action classes 30
9. Property condition classes. 32
10. Error helper classes 36
11. Data Provider Property categories 40
12. POSAppEventListener events and corresponding listener pairs 46
13. Data provider event listener proxies 49
14. POS Exception Log Entry data 59
15. Log levels with sample log entries 60
16. Exception types and response action categories 60
17. Client action descriptions 61
18. Functions of POSBC XML/Java messaging 62
19. Required soeps header fields. 64
20. Extending jifsrvc properties bundle 66
21. userbclistener.properties file 66
22. Standard Java properties file: posbc.properties 79
23. Failover types and solutions 80
24. Partial failover scenarios 81
25. Ownership stamps 85
26. States of a primary and backup controller 86
27. Development machine directory hierarchy 94
28. Request topics (1) 138
29. Request topics (2) 138
30. Error logging levels 157
31. Log levels with sample log entries 158
32. JAR commands 175
33. Payment system requirements 1 of 7 183
34. Payment system requirements 2 of 7 184
35. Payment system requirements 3 of 7 184
36. Payment system requirements 4 of 7 184
37. Payment system requirements 5 of 7 184
38. Variables for payment system requirements 5 of 7 185
39. Payment system requirements 6 of 7 185
40. Payment system requirements 7 of 7 185
41. SA user exit event numbers 193
42. Data Provider authorization properties (common) 212
43. Data Provider authorization properties (GSA) 213
44. Data Provider authorization properties (SA and ACE) 214
45. <cashReceipt> event 235
46. <coupon> event. 237
47. <dataEvent> event 239
48. <dataEvent> values 239
49. <transactionDiscount> event 240
50. <lineItemDiscount> event 241
51. itemID 243
52. <signOn> event. 247
53. Operator authorization attributes 247
54. <signOff> event. 249
55. <lock> event 249
56. <unlock> event 249
57. <customer> event 249
58. <options> event. 253
59. <points> event 259
60. <report> event 260
61. <scale> event 260
62. <tender> or <tenderDeclined> event 261
63. <transactionStart> event 263
64. <transactionVoid> event 265
65. <transactionSuspended> event 267
66. <transactionEnd> event 268
67. <transactionUpdate> event 269
68. <transactionTaxChange> event 269
69. <transactionTotals> event 270
70. <workstationStatus> event 271
71. <TransactionReplayComplete> event 271
72. Printer codes supported by Store Integrator 287

Safety

Before installing this product, read the Safety Information- Read This First.

قبل تركيب هذا المنتج، يجب قراءة الملاحظات الآمنية

Antes de instalar este produto, leia as Informações de Segurança.

在安装本产品之前，请仔细阅读 Safety Information
(安全信息)。

安裝本產品之前，請先閱讀「安全資訊」。

Prije instalacije ovog produkta obavezno pročitajte Sigurnosne Upute.

Před instalací tohoto produktu si přečtěte příručku bezpečnostních instrukcí.

Læs sikkerhedsforskrifterne, før du installerer dette produkt.

Lees voordat u dit product installeert eerst de veiligheidsvoorschriften.

Ennen kuin asennat tämän tuotteen, lue turvaohjeet kohdasta Safety Information.

Avant d'installer ce produit, lisez les consignes de sécurité.

Vor der Installation dieses Produkts die Sicherheitshinweise lesen.

Πριν εγκαταστήσετε το προϊόν αυτό, διαβάστε τις πληροφορίες ασφάλειας
(safety information).

.לפני שתתקינו מוצר זה, קראו את הוראות הבטיחות

A termék telepítése előtt olvassa el a Biztonsági előírásokat!

Prima di installare questo prodotto, leggere le Informazioni sulla Sicurezza.

製品の設置の前に、安全情報をお読みください。

본 제품을 설치하기 전에 안전 정보를 읽으십시오.

Пред да се инсталира овој продукт, прочитајте информацијата за безбедност.

Les sikkerhetsinformasjonen (Safety Information) før du installerer dette produktet.

Przed zainstalowaniem tego produktu, należy zapoznać się
z książką "Informacje dotyczące bezpieczeństwa" (Safety Information).

Antes de instalar este produto, leia as Informações sobre Segurança.

Перед установкой продукта прочтите инструкции по
технике безопасности.

Pred inštaláciou tohto zariadenia si pečítaje Bezpečnostné predpisy.

Pred namestitvijo tega proizvoda preberite Varnostne informacije.

Antes de instalar este producto, lea la informaci¾n de seguridad.

Läs säkerhetsinformationen innan du installerar den här produkten.

About this guide

This guide shows you how to integrate or extend point-of-sale (POS) applications using the Toshiba Global Commerce Solutions Application Extension Facility (AEF).

Who should read this guide

This guide is intended for programmers who are interested in using AEF to develop extensions to POS applications, or in using AEF to integrate Java™ programming and other functionality with existing POS logic. Those responsible for planning the installation and use of AEF, using the Store Integration Framework's system management facilities, and interested in customizing the system through the configuration process should consult the *Store Integrator User's Guide*. If you plan to customize the Store Integrator Graphical User Interface (SI GUI), then refer to *4690 Store Integrator Graphical User Interface Programming Guide*.

A working knowledge of the POS application in use, such as 4690 General Sales Application (GSA), Supermarket Application (SA), or Toshiba SurePOS™ Application Client/Server Environment for 4690 OS (ACE), the programming language used by that application (C-BASIC or C++), XML and the Java programming language, and of the Toshiba 4690 Operating System, is recommended.

How to use this guide

The documentation for programming the AEF consists of the following components:

- This document (*Store Integrator Programming Guide*).
- A JavaDoc version of the AEF application programming interface (API). JavaDoc is a standard HTML format for documenting a Java API. Although the JavaDoc for the AEF can stand on its own as an API reference, this guide also contains hyperlinks into the AEF JavaDoc.
- Code samples. This guide uses code samples to illustrate AEF API usage. Rather than include entire code modules in this guide, you can use the hyperlinks in this Programmers Guide to look at the code samples.

Using the hyperlinks

The JavaDoc, and code samples are included in the zip archive provided with this product. For the hyperlinks to work properly, the zipped files must be extracted into the root directory (C:\) of your file system. The folder (`siprogrammersguide`) created by the extract must be renamed `si`. The relative directory structure within the zip archive must be preserved.

Note: The JavaDoc and the code samples (Programmers Guide files) can also be downloaded from the Web using the same instructions as that of the Programmers Guide.

You must download the Programmers Guide pdf from the Web into the root directory.

To download the Programmers Guide go to: www.toshibacommerce.com/support. In the Software drop down menu select **Store Integration Framework and Data Integration Facility product support**, choose **SI Publications** under **Tips, FAQs and Publications**, and then select **Store Integrator**under the Store Integration Framework section. Open the latest Programmers Guide, click **File**>**Save As**, choose the directory, click **Open** and then click **Save**.

The directory must contain the elements as shown by the arrows in Figure 1. After the files are in the correct directory, open the Programmers Guide pdf and scroll to a code sample hyperlink. Click on the code sample hyperlink to open the code sample.

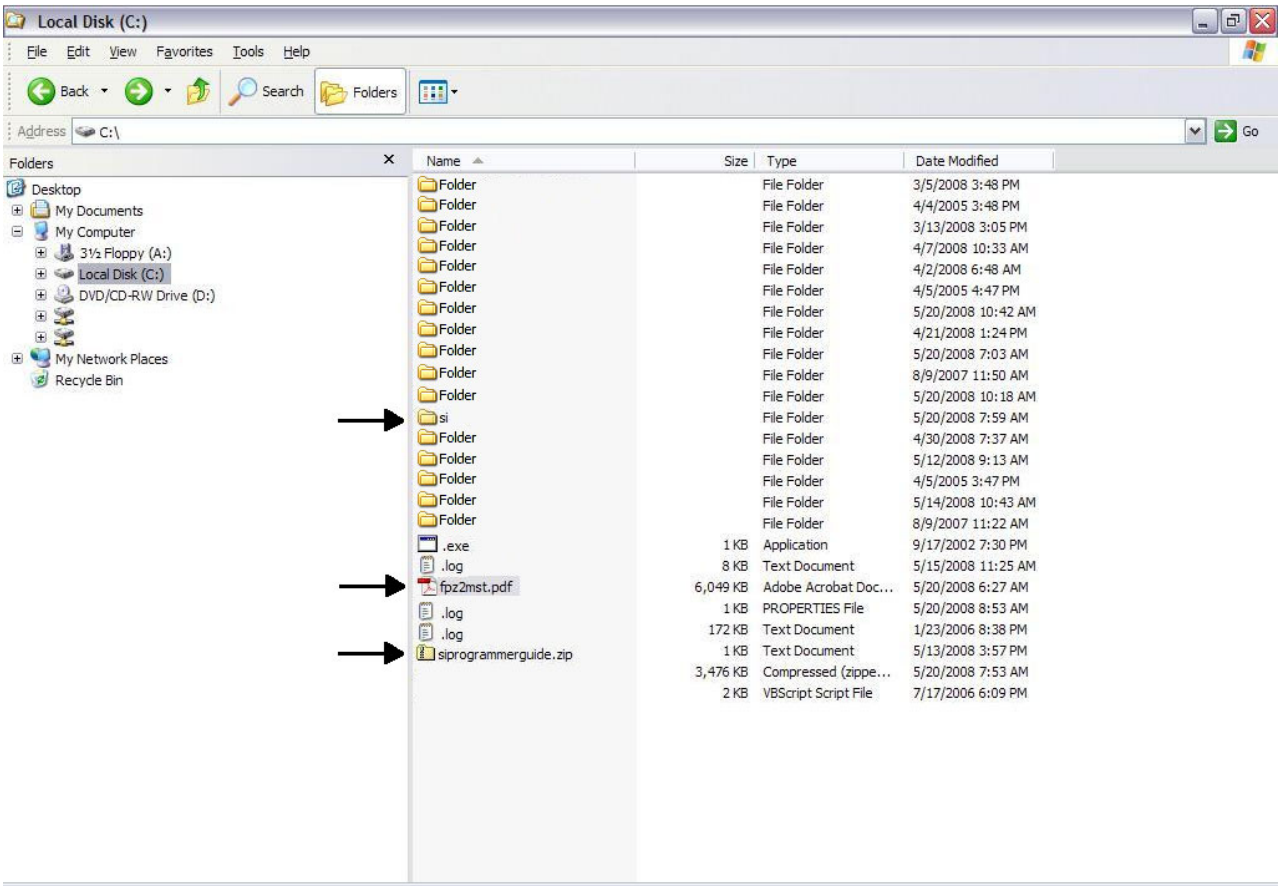
Local Disk (C:)
File Edit View Favorites Tools Help
Back Search Folders
Address C:\
Go
Folders
Desktop
My Documents
My Computer
3½ Floppy (A:)
Local Disk (C:)
DVD/CD-RW Drive (D:)
My Network Places
Recycle Bin
Name Size Type Date Modified
Folder File Folder 3/5/2008 3:48 PM
Folder File Folder 4/4/2005 3:48 PM
Folder File Folder 3/13/2008 3:05 PM
Folder File Folder 4/7/2008 10:33 AM
Folder File Folder 4/2/2008 6:48 AM
Folder File Folder 4/5/2005 4:47 PM
Folder File Folder 5/20/2008 10:42 AM
Folder File Folder 4/21/2008 1:24 PM
Folder File Folder 5/20/2008 7:03 AM
Folder File Folder 8/9/2007 11:50 AM
Folder File Folder 5/20/2008 10:18 AM
si File Folder 5/20/2008 7:59 AM
Folder File Folder 4/30/2008 7:37 AM
Folder File Folder 5/12/2008 9:13 AM
Folder File Folder 4/5/2005 3:47 PM
Folder File Folder 5/14/2008 10:43 AM
Folder File Folder 8/9/2007 11:22 AM
.exe 1 KB Application 9/17/2002 7:30 PM
.log 8 KB Text Document 5/15/2008 11:25 AM
fpz2mst.pdf 6,049 KB Adobe Acrobat Doc... 5/20/2008 6:27 AM
.log 1 KB PROPERTIES File 5/20/2008 8:53 AM
.log 172 KB Text Document 1/23/2006 8:38 PM
siprogrammerguide.zip 1 KB Text Document 5/13/2008 3:57 PM
3,476 KB Compressed (zippe... 5/20/2008 7:53 AM
2 KB VBScript Script File 7/17/2006 6:09 PM

Figure 1. Example of file directory

Where to find more information

Current versions of Toshiba Global Commerce Solutions publications are available on the Toshiba Web site at http://www.toshibacommerce.com.

1. Select **Support**
2. Select **Publications**.

The following publications provide additional information:

- Data Integration Facility User Guide (GC30-4077)
- Remote Management Agent and Viewer User's Guide (GC30-4106)
- 4690 Store Integrator Graphical User Interface Programming Guide (G362-0563)
- Store Integration Framework white paper (G581-0262)
- Store Integrator User's Guide (G362-0561)
- Retail on demand brochure (G580-3980)
- SurePOS Application Client/Server Environment for 4690 OS Programming Reference (GC30-4150)
- Supermarket Application Programmers Guide (GC30-3634)
- General Sales Application Planning and Installation Guide (GC30-3690)

Notice statements

Notices in this guide are defined as follows:

Notes These notices provide important tips, guidance, or advice.

Important
These notices provide information or advice that might help you avoid inconvenient or problem situations.

Attention
These notices indicate potential damage to programs, devices, or data. An attention notice is placed just before the instruction or situation in which damage could occur.

CAUTION
These statements indicate situations that can be potentially hazardous to you. A caution statement is placed just before the description of a potentially hazardous procedure step or situation.

DANGER
These statements indicate situations that can be potentially lethal or extremely hazardous to you. A danger statement is placed just before the description of a potentially lethal or extremely hazardous procedure step or situation.

Summary of changes

June 2014

This edition includes updates for Store Integrator Version 3 Release 5.

February 2013

This edition includes updates for Store Integrator Version 3 Release 4.

October 2011

This edition includes updates for Store Integrator Version 3 Release 3.

Chapter 1. Overview

The Toshiba Store Integrator (SI) delivers the next generation store platform. Store Integrator provides the foundation for integrating existing store solutions, adding new solutions and touch points, and integrating the store to the Enterprise. Unlike the typical store configuration, this solution is a fully-integrated environment based upon industry standards. Store Integrator supports the integration of Toshiba, Business Partner POS solutions, and hardware from various providers.

The Toshiba Store Integrator helps retailers with the current challenges:

- Difficult access to real-time information
- High maintenance costs
- Complex and expensive integration
- Custom-made solutions that are highly dependent upon the stability of the provider

Store Integrator has been designed to easily facilitate Point of Sale (POS) application integration with various other client applications. Applications can be integrated through a Java interface, and an XML API. By integrating with existing POS applications, client applications are able to monitor and control existing POS logic.

The Toshiba Application Extension Facility (AEF) is a component of the Store Integrator (SI). The AEF is a Java application programming interface (API) that allows client code to observe and interact with working POS sessions. To understand the use of the AEF API, it is necessary to understand some of its objectives and architecture.

- Programmatic access to POS functions using a Java API.
  - Use of the API should not require extensive retail domain knowledge.
  - The API should allow remote access.
  - The API should provide a level of abstraction from the POS application such that client code can be somewhat independent from the POS application.
- Integration of POS logic into other applications. Rather than attempt to duplicate complicated POS application logic, the objective is to provide access to existing application logic. This is done by allowing the POS application to run in a virtual session. An example of integrating with POS logic is a Web-based home shopping application. The home shopping application is implemented as a Servlet that uses virtual POS sessions from the customer's home store. Because the Servlet is using the same POS application logic, the customer using the browser-based home shopping solution is provided the same pricing information as a register in the store would provide.
- Extension of the POS application through Java. The same Java API that applies to virtual POS register sessions can be applied to real POS registers. This allows users to write register extensions in Java code. An example of this type of extension is integrating a radio frequency ID (RFID) reader with a POS register. The AEF API can be used to cause a new operator key sequence to trigger the RFID reader to scan the contents of a basket. The AEF API can then be used to inject the items into the transaction on the POS register.
- Remote management functions consisting of monitoring and controlling the real and virtual POS sessions and their environments.

The POS Business Component (POSBC) interface extends the capabilities of AEF by supporting an XML-based interface to a POS application. Thus, the flexibility of SI is increased since the XML interface is not limited to only Java clients.

AEF architecture overview

One of the AEF objectives is to provide a standard Java interface to multiple POS applications. For this release of Store Integrator, the following applications support the AEF API:

- 4680-4690 General Sales Application (GSA)
- 4680-4690 Supermarket Application (SA)
- SurePOS Application Client/Server Environment for 4690 OS (ACE)

Within the basic AEF architecture, the POS terminal sales application is AEF-enabled with a set of hooks that generate XML events as illustrated in Figure 2. These events indicate the state of the POS application. Events are generated for POS actions such as operator logon and logoff, start or end of transaction, item sale, tender, and so on.

The AEF uses these events to maintain a Java object model representing the state of the POS application. Each instance of the POS terminal sales application is wrapped by an instance of AEFSession. The session object and the Java objects contained within the session together form the Java AEF API, which you can use to interact with the POS logic. The AEF API can also drive the POS terminal sales application by generating device input such as keyboard or scanner input.

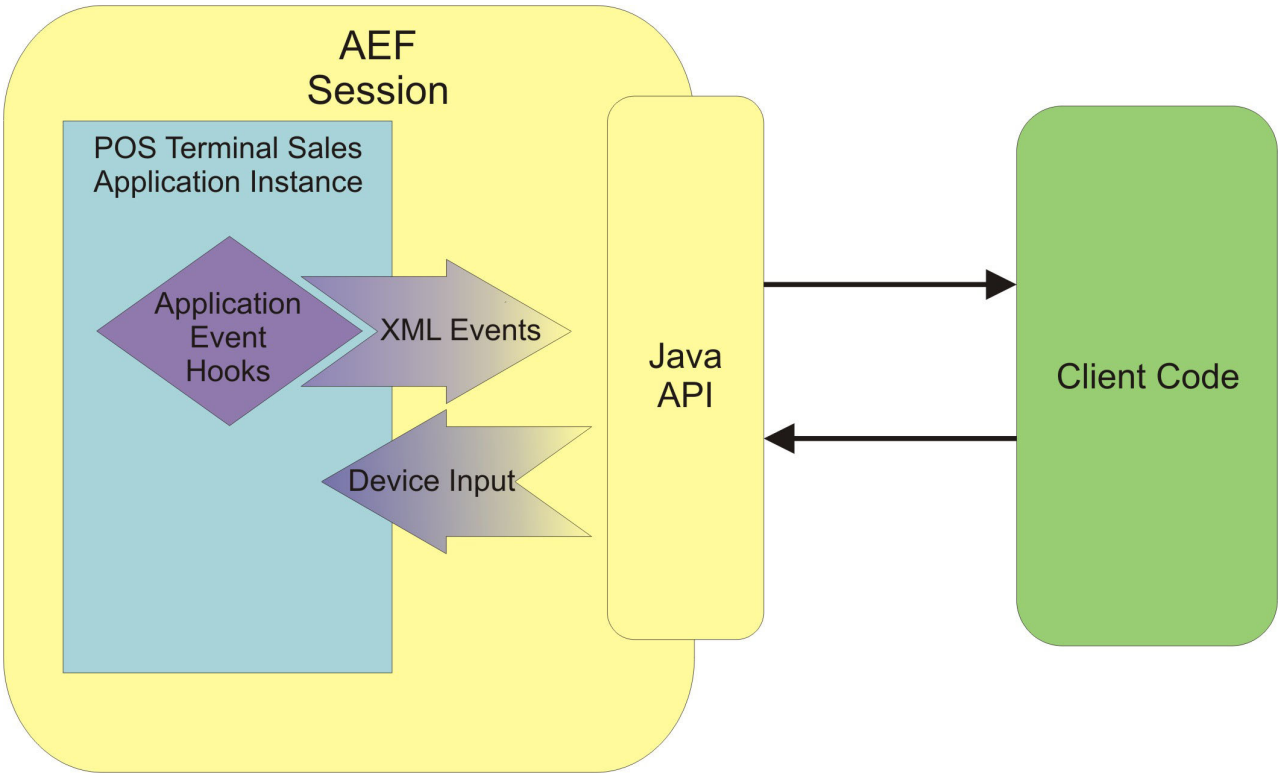

Figure 2. AEFSession architecture

Real or virtual sessions

AEF sessions can be configured to run on a physical 4690 register, or on a virtual session within the Client Session Server (CSS) environment. To run on a physical 4690 terminal, the terminal must be configured to run the terminal sales application as well as the AEF. In this configuration, the client Java code typically runs inside the same terminal Java Virtual Machine (JVM) as the AEF. Figure 3 on page 3 shows the AEF running in a physical 4690 terminal, with the client code co-resident within the same JVM. Because the SI GUI is a client of the AEF API, it is an example of this configuration when running

on a 4690 register.

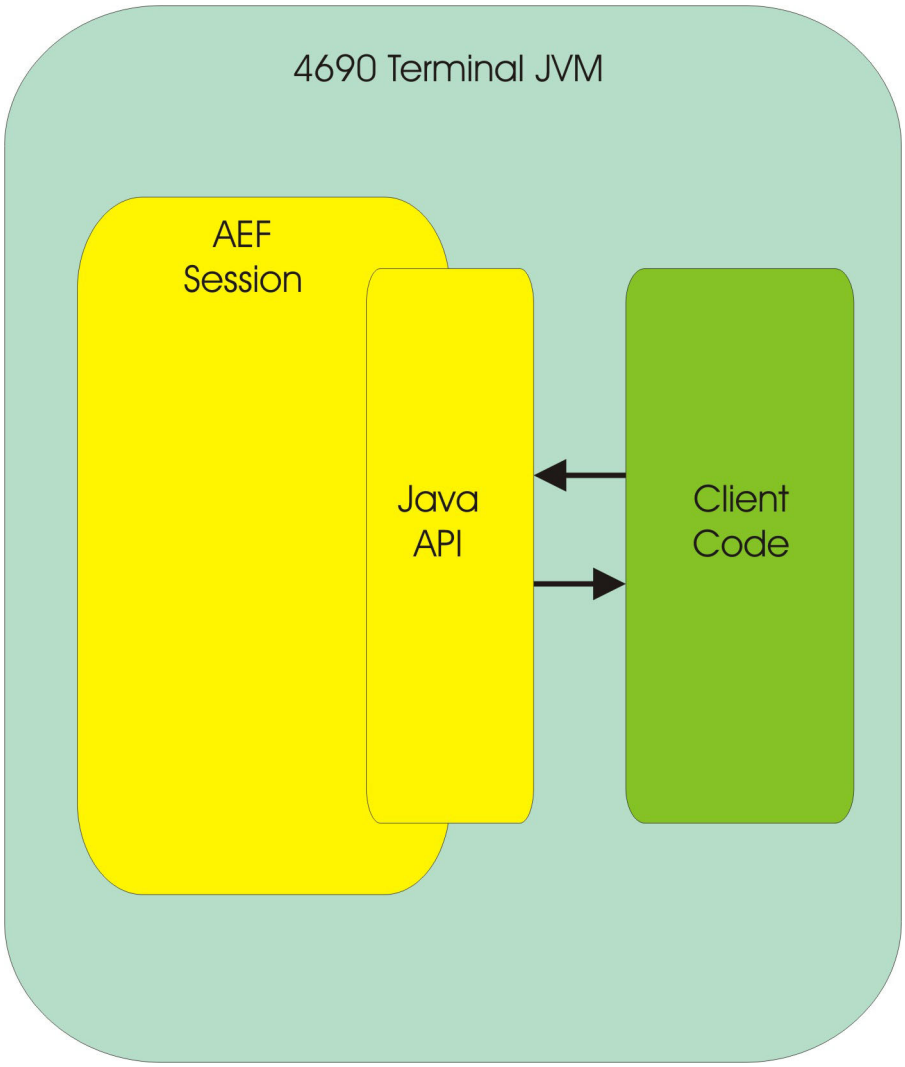

Figure 3. 4690 terminal session

Virtual AEF sessions can be configured to run on 4690 controllers. In this configuration, the same terminal sales program is run on the 4690 controller as the one which runs in the physical registers, with the exception that all I/O devices are emulated. A separate Java AEFSession object wraps each instance of the terminal sales application. The client (which is typically remote) connects with one or more virtual sessions within the CSS JVM. Figure 4 on page 4 illustrates CSS with multiple virtual sessions.

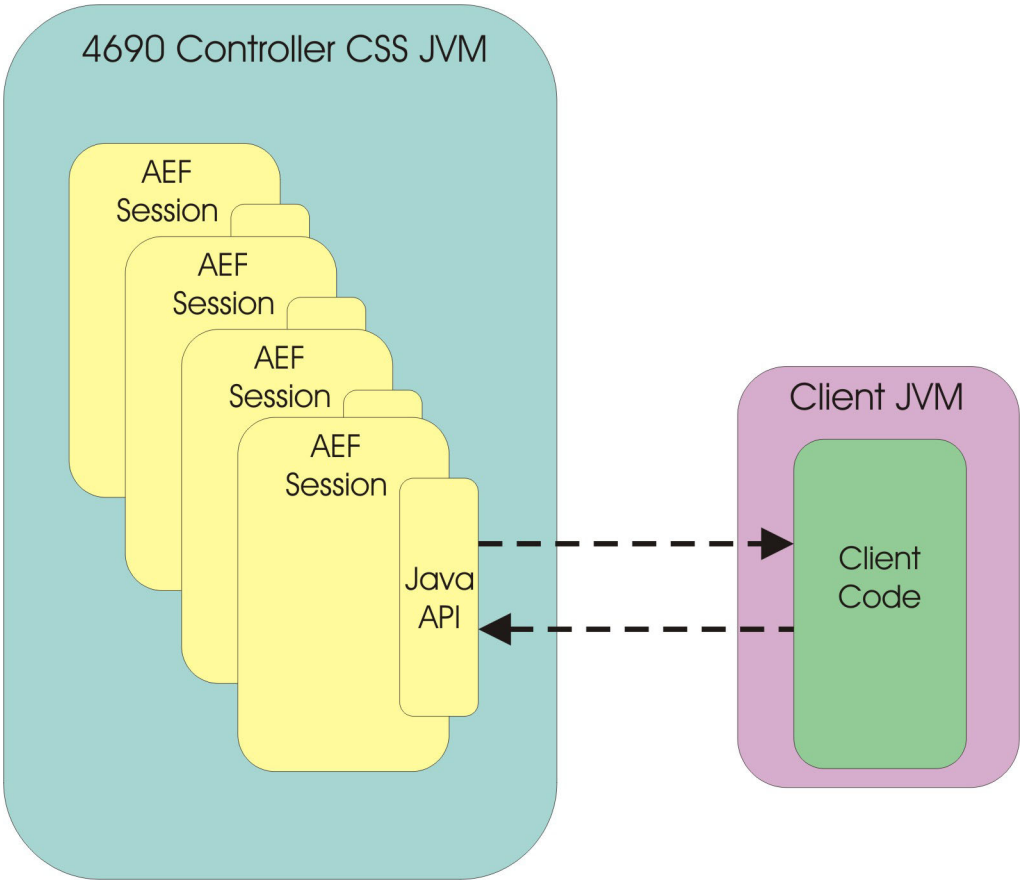

Figure 4. 4690 CSS sessions

AEF programming tasks

The following list is a summary of Store Integrator AEF programming tasks:

- Setting up the development environment
- Using the Quick Start example
- Getting a session server
- Connecting to a specific session server
- Getting a specific session
- Getting a session without a Ready Wait
- Retrieving a data provider property value
- Subscribing to a data provider event
- Sending additional data to an event
- Sending a new XML event from the POS application
- Using an AEF session
- Extending existing Automation Provider APIs
- Extending the Automation Provider API with a new function
- Using error logging
- Writing an error helper
- Handling exceptions

- Using properties file configuration
- Packaging user code and properties files
- Using the 4690 Debugger for POS applications under CSS

POSProvider/POSServer API overview

The POSProvider/POSServer API is an enhancement to the AEF that provides a client interface for sending requests to, and receiving results from, a point-of-sale (POS) system. In addition to the basic POS functions provided by the AEF, the POSProvider/POSServer API allows you to:

- Check the status of the point-of-sale system
- Request unit price information
- Request extended prices and transaction totals for an order
- Process and pay for an order
- Use Checkpointing for payment integrity by using a simple protocol to determine whether the client or the server is responsible for refunding a charge to the customer. The answer is then logged so a manual correction can be made. Automatic refunds are not supported.
- Failover to a backup controller if the primary is unavailable
- Reduce the network traffic by reducing the amount of communication between the client application and the POS system. The client and POS applications interact through composite requests and results instead of individual commands.

Note: The POSProvider has its own failover function and therefore cannot be used with AEF failover-enabled sessions.

POSProvider/POSServer architecture overview

Figure 5 on page 6 shows an overview of the major system components of the POSProvider/POSServer API. The left side of the figure represents the client system. The right side of the figure is the traditional Toshiba POS controller system. The client interacts and communicates with the controller system through a network using Java Remote Method Invocation (RMI) technology.

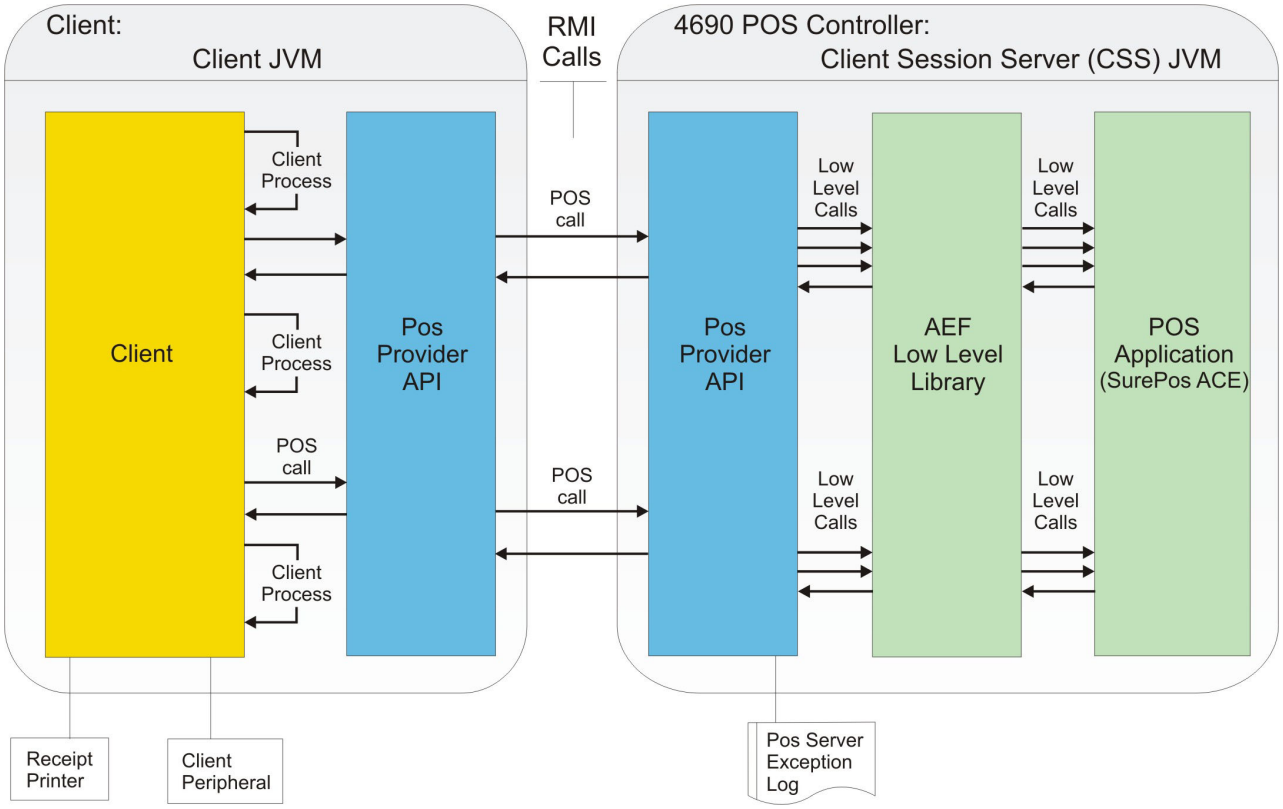

Figure 5. Major system components for POSProvider/POSServer

Point of Sale Business Component overview

The POS Business Component (POSBC) interface extends the capabilities of AEF by providing a standard, XML-based messaging interface. The objective of the POSBC is to support POS integration by allowing access by multiple types of clients (fuel, portable shopper, and others) in multiple connectivity and programming environments. The POSBC defines a standard interface to POS functionality through an XML schema and provides adapter tooling that facilitates the implementation of that interface for both Toshiba and non-Toshiba POS applications. For more detailed information about the XML messages, the XML schemas can be located in the %AEF_HOME%\schema directory of your installation environment.

POSBC provides the following functional categories:

- Control and sales-related
- Logical printer
- Logical receipt

POSBC control and sales-related functions

The clients of the POSBC require several control functions to use the API:

Initialize

Clients will begin by initializing a POSBC session to bring it to a known state where further actions can be performed. During this request, if the POSBC configuration has devices enabled, the devices will also be initialized.

Terminate
Allows clients to terminate POSBC sessions to allow offline processing to occur (such as store close).

ResetTerminal
Used to return the POSBC and POS application to a state where a new transaction can begin.

Signon and Signoff
Support the concept of a "Closed" state.

QueryStatus
Used to determine the state of the POSBC and to get a limited history of various data events such as a list of operator prompts or a list of API functions performed.

Note: This function might return escaped special characters when reporting the history of POSBC API requests and responses. Client applications are expected to parse these characters properly. The POSBC does not format these characters for logging purposes. Therefore, users reading POSBC log files should expect to see these characters when viewing QueryStatus responses.

The sales-related functions are the core of the POSBC functionality. The POSBC API is designed to provide access to POS functions so that the following typical sales-related actions are supported:

AddItem
Add an item to the current sales transaction, starting a sales transaction, if necessary. Various attributes of the item may be provided (such as weight, price, tare and others). Item codes can be qualified as being either key-entered or scanned.

VoidItem
Remove an item from the current sales transaction.

AddCoupon
Add a coupon to the current sales transaction.

AddCustomerNumber
Add a customer (loyalty) number to the current sales transaction.

GetTotals
Get totals information about the current sales transaction.

Suspend
Suspend the current sales transaction.

AddTender
Add a tender to the current sales transaction. The tenders supported are cash, credit, debit, EBT Cash, and EBT Food. For tenders that require a pin pad, a physical pin pad must be configured, online, and used to input required data (such as the PIN) for those tenders to be added to a transaction.

POSBC logical printer functions

In the POSBC environment, the POS printer is a device that is owned and managed by the POSBC, and the POSBC XML clients can use it. To support this configuration, the POSBC provides a "logical" printer API that can be used by POSBC clients to manage and print data on the POS printer using a fairly high-level language: Printer Markup Language (PML). The printer operations that can be performed by POSBC clients are:

InitializePrinter
This request configures the virtual printer if it is not already initialized.

Print This function can take in a PML string, a URL to a complete PML document, or a list of FormattedReceiptLineObjects to print.

Note:

1. For a printer to be supported logically by POSBC, JavaPOS drivers must exist and be available for it.
2. Check Franking on the document insert (DI) station is supported, however other DI station functions (such as MICR reading, Check Image Scanning, and Check Face Printing) are not supported.
3. Support for Fiscal printers is not provided.

POSBC logical receipt functions

The POSBC supports the concept of logical "receipts," which contain information about the current POS transaction. For example, a logical sales receipt can be used to support an electronic scrolling receipt on a Personal Shopper display.

POSBC clients can manage their transaction receipts with these functions:

- AddReceiptLines
- RemoveReceiptLines
- RemoveReceipts
- ArchiveCurrentReceipts
- PrintCurrentReceipts
- PurgeArchivedReceipts

Note:

1. If *clean receipt* functionality is used by the POS application, then some restrictions apply since POSBC has limited support for it. For example, if the POS application is using clean receipt, then the POSBC client should only add lines to the end of the logical receipt.
2. Under certain circumstances, the receipt needs to be able to be "cleaned up". The only scenario where clean up is supported is when an item is voided from the transaction and both the item void lines, as well as the original item sale lines need to be removed from the receipt. This behavior is not executed for all item void requests to the POSBC; the client needs to indicate in the void item request whether the receipt lines are cleaned up.

POSBC architecture overview

Figure 6 on page 9 explains the POSBC architecture.

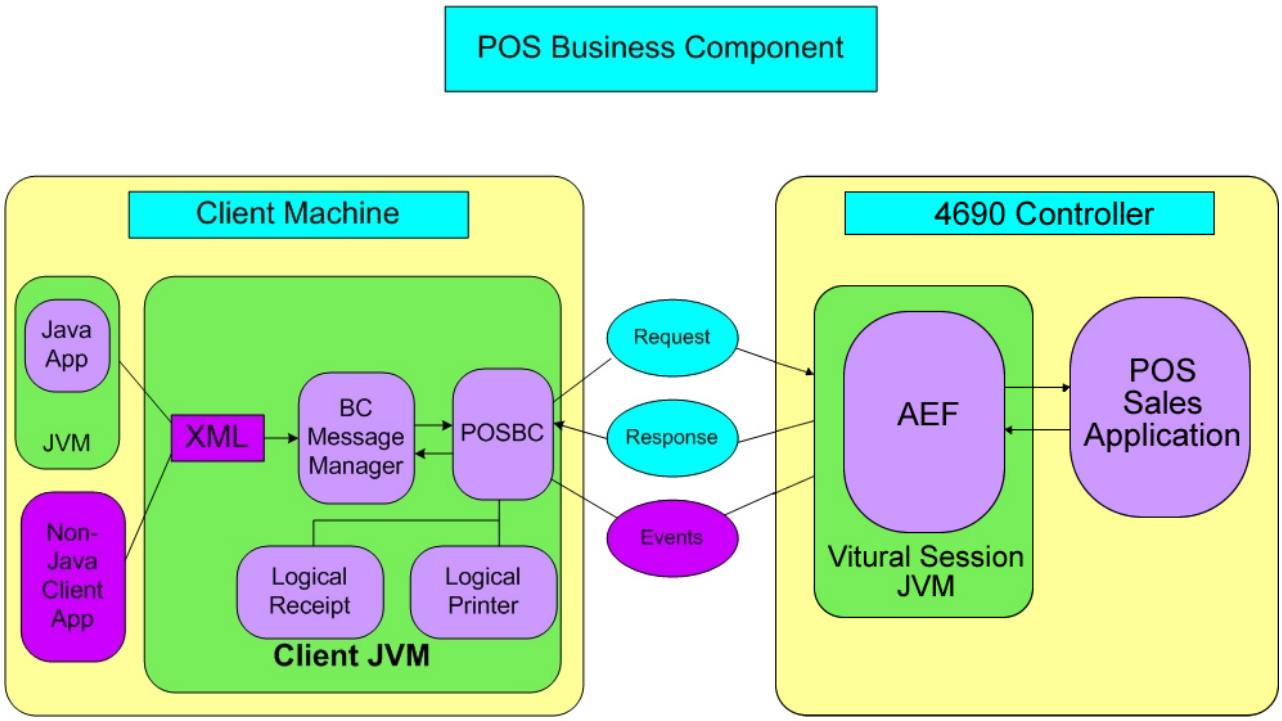

Figure 6. POSBC architecture

POS Sales Application

The POS Sales Application is the individual retailer-specific version of the Toshiba Global Commerce Solutions ACE, SA or GSA sales application. This application has been AEF-enabled to work with the Toshiba SI Application Extension Facility (AEF) product. See "Enabling user extensions and additional features" on page 186.

In many cases, the POS application will have modifications that provide unique retailer business functionality. Depending on the relevance to the client application, these might or might not need to be exposed to the client through extensions to the POSBC and SI AEF systems.

AEF API

The AEF is the set of application programming interfaces (APIs) provided by the Toshiba Global Commerce Solutions SI product.

POSBC

POSBC is the XML client interface for sending requests to a point-of-sale (POS) system. POSBC includes the basic set of operations to perform the tasks required for order processing through the POS system.

Known Issues and Limitations

- OS 4690 Enhanced Mode Java RMI - There is a limitation in the OS 4690 Enhanced mode operating system networking stack that will not allow Java RMI communications to occur between the Java 1.4 JVM and the Java 1.6 JVM on the same system. For example, this limitation will prevent custom solutions from running an application in the Java 1.6 JVM which communicates via RMI with the Client Session Server (Java 1.4 JVM) on the same controller.
- Terminal Transfer limitation for POSBC Client - If a terminal transfer is attempted for a POSBC session where the operator or transaction is transferred from POSBC to another terminal or session, the POSBC

session remains in an unknown state as the transaction/operator cleanup is not communicated to the POSBC client. POSBC session cleanup then requires full reinitialization of the client. Recovery should be set to *false* on the succeeding InitializationRequest.

Chapter 2. Understanding the architecture

The AEF API is divided into the categories shown in Table 1.

Table 1. AEF API categories

Category	Description
Infrastructure	Includes classes that support creating and remotely accessing sessions.
Automation Provider	A set of classes that provides an object model of the POS application and transactions. Includes synchronous operations such as starting transactions, adding items and tenders. This API drives the POS application.
Data Provider	The data provider monitors events and provides information.
Device Accessor	Provides control of the POS I/O. Allows insertion of client device hooks that can observe, consume, and/or inject device I/O.
Service Provider	Provides an API developers can use to write services that interact with the point of sale application.

Infrastructure

The following sections describe the primary infrastructure components of the AEF. These components are responsible for creating sessions, and providing remote access to them.

AEFSession

The AEFSession encapsulates access to a POS application running on a real register or a virtual session under the control of CSS. In the 4690 AEF environment, each terminal (real or virtual) has an instance of AEFSession that supports both local and remote access. The AEFSession object works remotely through Java Remote Method Invocation (RMI). The AEFSession exposes three API sets as described in the following sections (see Figure 7 on page 12).

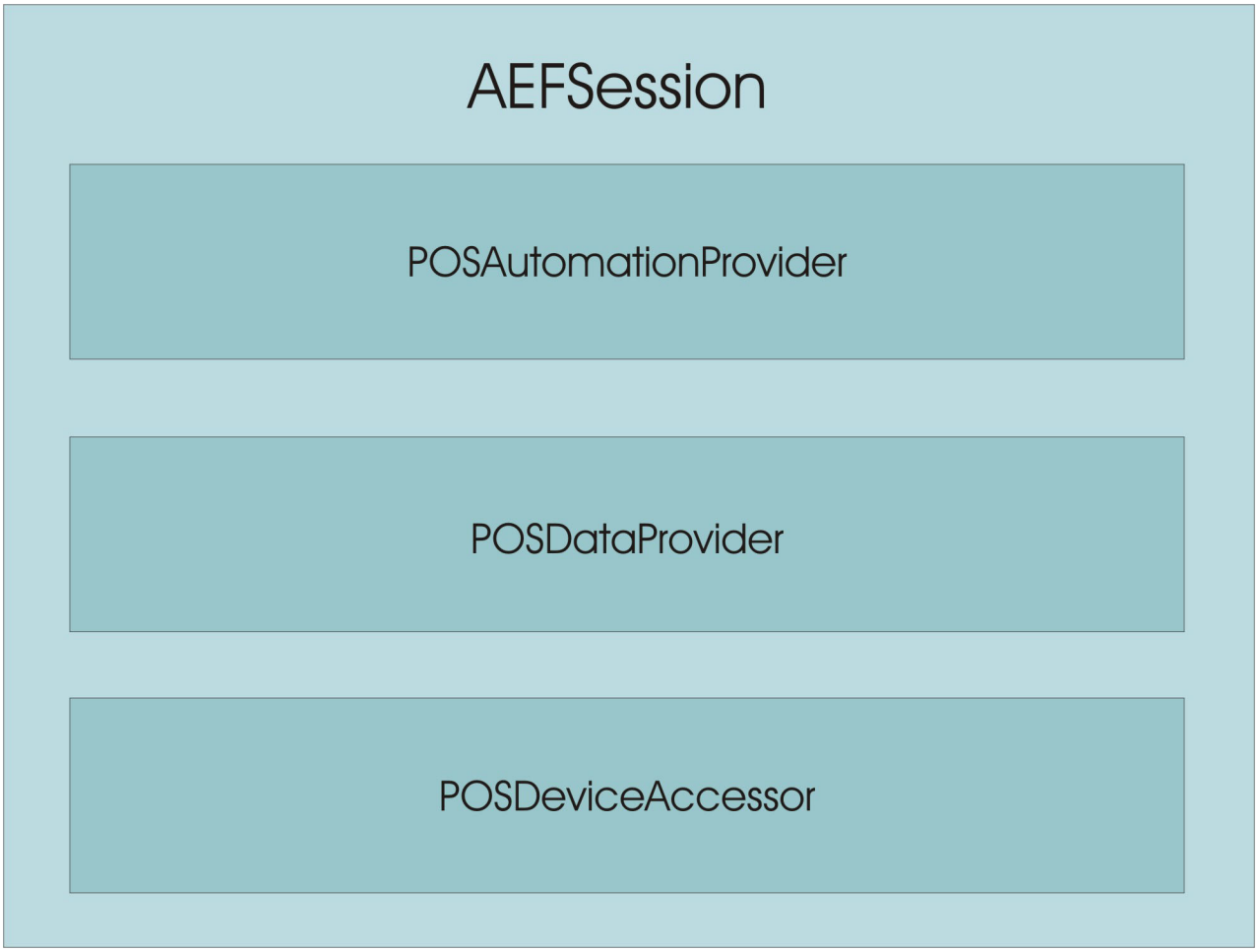

Figure 7. AEFSession components

POSAutomationProvider

The POSAutomationProvider API can be used to actively drive the session. Examples include starting a transaction, and adding items and tenders to the transaction. This API is exposed through a Java object model that represents the transactions and line items for the register. The POSAutomationProvider API provides a certain level of POS application isolation so that common functions can be executed using the POSAutomationProvider API regardless of the underlying POS application.

POSDataProvider

The POSDataProvider API provides a mechanism for passively observing events occurring in the underlying POS application. This API supports both local and remote access.

POSDeviceAccessor

The Device Accessor API provides a JavaPOS interface for observing, modifying, injecting, and consuming POS I/O device information. The API allows hooks to be installed between the POS application and the POS I/O devices.

AEFSessionFactory

The AEFSessionFactory provides access to AEFSession instances. On a real register, the AEFSessionFactory is able to return the single instance of AEFSession for that register. An AEFSessionFactory running in the CSS JVM is able to use CSS to start new virtual instances of the POS application. The AEFSessionFactory can return pre-existing virtual sessions, or create new ones as needed.

Each factory maintains a pool of available sessions. Session factories can only exist on machines running Toshiba 4690 OS. In a typical store, there is an AEFSessionFactory instance in each real register, as well as an instance in each controller JVM running CSS.

Client code does not need to interact directly with the AEFSessionFactory. Instead, it uses the AEFBase class to gain access to a session. The AEFBase class contacts a SessionServer on the client's behalf. The SessionServer then determines the appropriate AEFSessionFactory to handle the request.

Depending on AEF configuration options, the AEFSessionFactory can be exported through RMI. The factory ***must*** be exported if remote access to the sessions owned by the factory is required. By default, the factories on the 4690 controllers and 4690 terminals are exported. Some processing time is involved in exporting a factory, so if remote access to the factory's sessions is not required, the AEF should be configured not to export the factory. For example, if remote access to terminal sessions is not required, you should configure the terminal factories as non-exported. If a factory is configured as exported, it periodically sends a beacon to inform any SessionServer instances of its location.

AEFSessionPool

Each AEFSessionFactory contains a session pool, implementing the AEFSessionPool interface, that maintains a collection of sessions managed by that factory. The sessions can be acquired by the factory, and released back into the pool at a later time.

SessionServer

The SessionServer handles requests for sessions. If the request is for a specific terminal number, the SessionServer determines which factory can provide the requested session. If the request is for any available session, the SessionServer uses a load balancer to determine which factory should serve the request.

Multiple SessionServer instances can be configured within the store. Every SessionServer should know about all the exported factories in the store. Therefore, any SessionServer should be able to handle a request for a session, regardless of which factory ultimately provides the session.

The SessionServer is designed to listen for AEFSessionFactories within the same Transmission Control Protocol/Internet Protocol (TCP/IP) subnet.

Figure 8 on page 14 illustrates how a session is retrieved using a SessionServer.

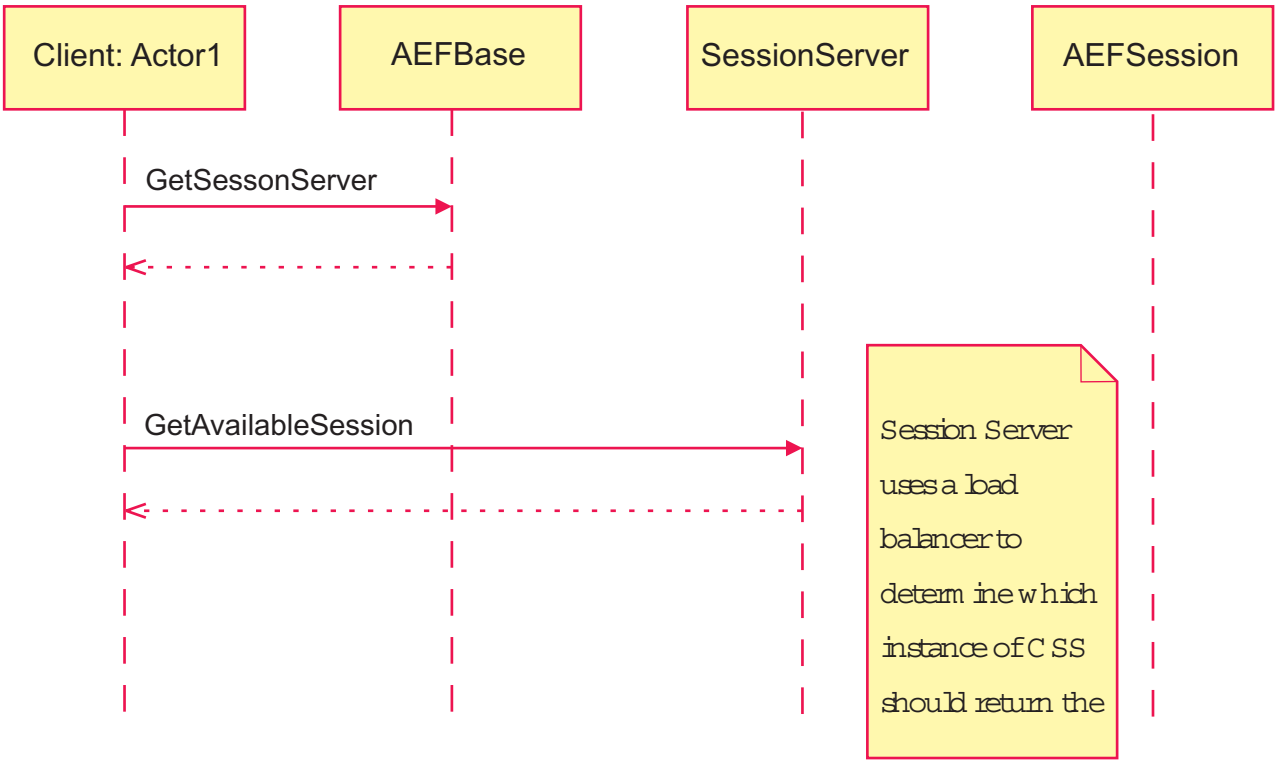

Figure 8. Retrieving a session using a SessionServer

LoadBalancer

Each SessionServer includes a LoadBalancer that helps determine which session factory should handle the next request for any available virtual session. Note that requests for real terminal sessions do not require load balancing. The default load balancing implementation uses round-robin allocation of session requests to each of the exported virtual session factories in the store.

The default load balancers have no knowledge of each other, so if there are multiple clients requesting available sessions from multiple session servers, the round-robin behavior only applies internally within each SessionServer, and not among all SessionServer instances.

The list of factories that the load balancer knows about is dynamic, as machines in the network may come online or be rebooted. This can cause what appear to be anomalies in the "round-robin" load balancing behavior, but is expected when a session server starts up or shuts down. If more sophisticated load balancing behavior is required, the default load balancer implementation may be replaced by a factory mechanism.

AEFBase

The AEFBase is a class that is the gateway to the AEF. All JVMs that either run the AEF, or are clients of the AEF create an instance of the AEFBase class. Depending upon the platform and configuration settings, the AEFBase may create and export instances of the SessionServer, the AEFSessionFactory, the DeviceServer (used for remote devices), the LoggerControl (used for setting logging levels), the DebugMemoryImpl (used to remotely query memory usage), and the RMI registry. The AEFBase contains methods for requesting a specific AEFSession by terminal number, and for requesting any available virtual session. This is the recommended method for clients to gain access to an AEFSession. The AEFBase instance contacts a SessionServer that uses its LoadBalancer to determine which AEFSessionFactory should handle the request for the session.

Remote Method Invocation (RMI) heartbeats

RMI heartbeats are multicast messages used at the socket layer to help determine if a remote machine has become disconnected from the network. The heartbeat messages are used to update a table that is queried when a socket read timeout occurs. If the entries in the table indicate that the remote machine is still active, then the read is retried (indefinitely).

To determine if the remote machine is active, the system compares the amount of time since the last heartbeat was received to three times the normal interval of when the heartbeat is expected. If the time since the last heartbeat is longer, the assumption is that the remote machine is no longer connected to the network and read timeout exceptions will be thrown.

On a remote machine, if the operating system is under heavy load and does not give a time slice to its heartbeat thread in a timely manner, then the heartbeat will arrive late. This heartbeat delay can lead to timeouts on any other remote machines interacting with this machine.

For serviceability, AEF logs performance warnings any time a heartbeat is sent out at a time that is more than twice the expected interval. An example of this warning is as follows:

WARNING: Performance Warning - Time since last RMI heartbeat exceeded twice the normal 4500 ms interval: 10546 ms
Figure 9. Warning example

If read timeouts are occurring frequently paired with heartbeat performance warnings due to a heavy system load, the RMI heartbeat interval time can be increased. This property resides in config.properties and controls how often (in milliseconds) nodes containing an instance of AEFBase will emit an RMI heartbeat. The default entry is:

```
rmi.heartbeat.interval=4500
```

AEFException

The AEF exception model defines a single exception class named AEFException that is thrown to indicate errors. This error is indicated using a primary error code, and an extended error code.

Component configurations

Different configurations of the components discussed here vary depending on the platform of execution. A typical in-store configuration is shown in Figure 10 on page 16.

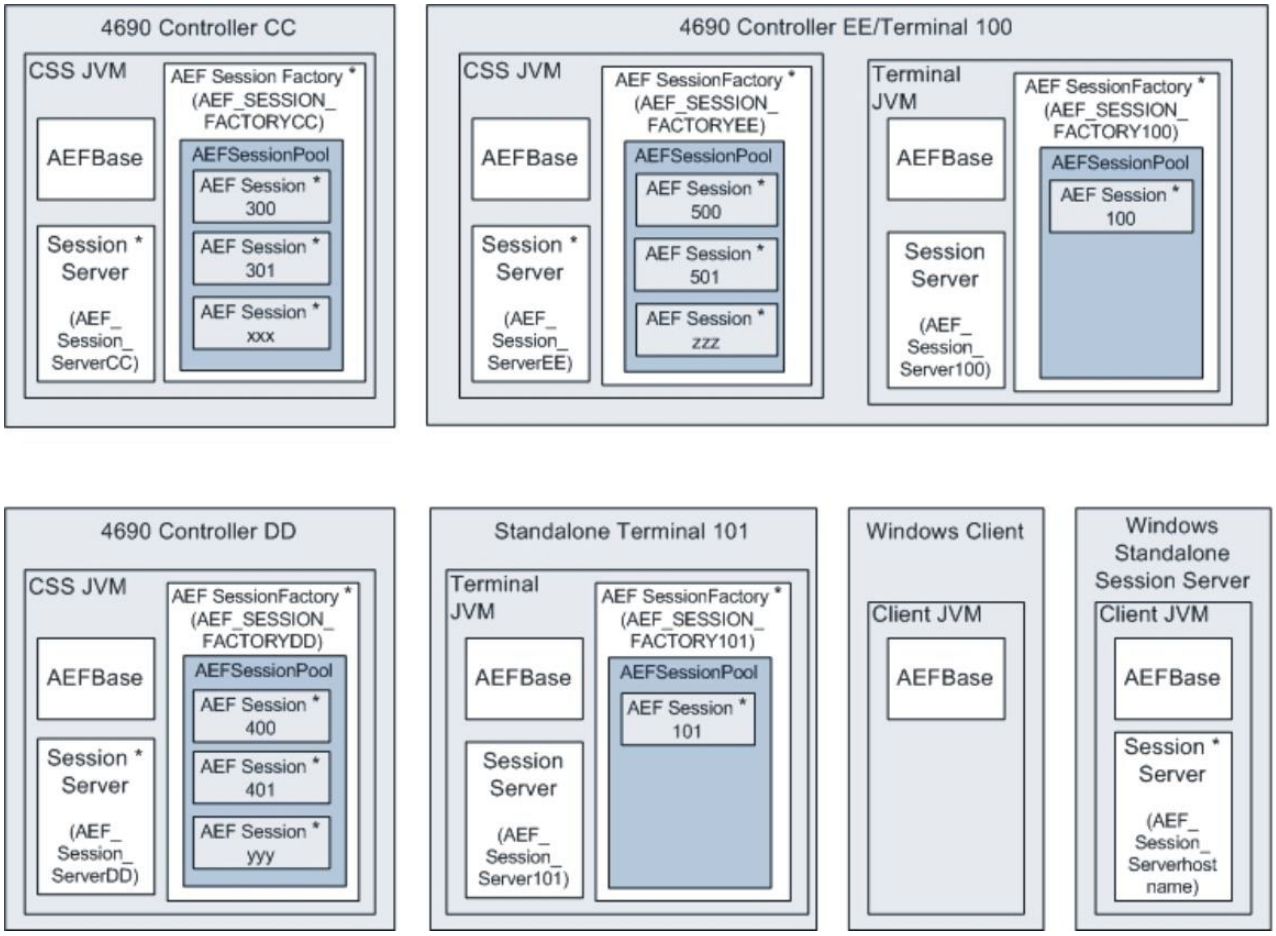

Figure 10. Typical in-store configuration

The asterisks in the upper right corners of the groups shown in Figure 10 indicate that the object is exported and remotely accessible.

Figure 10 illustrates a store that is using SI to provide access to both virtual and real terminal sessions. CSS is configured to run on the 4690 controllers CC, DD, and on the controller/terminal EE. Each of these controllers contains an exported session server, and an exported session factory. Each of the session factories includes a session pool that manages the sessions. As with real terminal numbers, the virtual terminal session numbers must be unique. They must not conflict with virtual terminal session numbers from other controllers, or with real terminal numbers in the store.

This example also demonstrates access to real terminal sessions. Terminal number 100 is the terminal portion of a 4690 controller/terminal, and terminal 101 is a standalone terminal. The session pools for real terminal sessions always contain the single terminal session. The session factories for real terminals are exported by default, allowing remote access to the terminal session. However, it is more likely that access to the real terminal sessions is local within the terminal JVM. For example, the SI GUI is a local client of the AEF (running in the same JVM as the AEF). If no remote access to the real terminal sessions is required, the AEF can be configured not to export the terminal session factories. This exclusion would result in a reduction of network processing overhead.

Figure 10 also shows a remote Windows client (running Java 1.13.3 or later) that needs to gain access to a virtual session. In this case the client creates an instance of AEFBase. The AEFBase instance may be used to gain a remote reference to a specific terminal session, or any available virtual terminal session. The lower right rectangle in the figure represents a standalone session server. In certain store environments, there may be an advantage in having a session server independent from the 4690 controllers. The session

servers maintain a list of all the factories in the store, and are capable of restoring factory connections if the factory machines are stopped and restarted. This means that a standalone session server could automatically restore references to the factories when an IPL is performed on the 4690 controllers and terminals. The client can continue to use the standalone session server, even after an IPL of the 4690 machines is performed.

Configuration properties

The AEF can be configured using properties that affect various behavior of the AEF. The following sections describe some of the properties that affect AEFBase and AEFSession instances.

Table 2 documents configuration properties that impact the AEFBase. The default setting may be dependent on the platform of execution.

Table 2. AEFBase configuration properties

Behavior or Value	Default on 4690 Controller	Default on 4690 Terminal	Default on Other Platform	Configuration Property to Override
AEFBase creates session server	True	True	False	userconfig.properties create.server=true | false
Session server is exported	True	False	N/A (Server not created by default)	userconfig.properties server.remote.access=true | false
Session server ID	AEF_SESSION_ SERVER*xx* where *xx* is the controller node ID.	AEF_SESSION_ SERVER*yyy* where *yyy* is the terminal number.	AEF_SESSION_ SERVER*hostname* where *hostname* is the TCP/IP hostname.	userconfig.properties server.id=
AEFBase creates session factory	True	True	N/A (Factory not allowed)	userconfig.properties create.factory=true | false
Factory is exported	True	True	N/A (Factory not allowed)	userconfig.properties factory.remote.access=true | false
Factory ID	AEF_SESSION_ FACTORY*xx* where *xx* is the controller node ID.	AEF_SESSION_ FACTORY*yyy* where *yyy* is the terminal number.	N/A (Factory not allowed)	userconfig.properties factory.id=
AEFBase creates RMI registry	True	True	False	userconfig.properties create.rmi.registry=true | false
RMI port	12099	11099	11099	userconfig.properties rmi.port=
AEFBase creates remotely accessible logger control	True	False	False	userconfig.properties logger.remote.access=true | false

Table 2. AEFBase configuration properties (continued)

Behavior or Value	Default on 4690 Controller	Default on 4690 Terminal	Default on Other Platform	Configuration Property to Override
AEFBase creates device server for exporting devices to remote client	False	False	False	userconfig.properties create.device.server=true | false
Device server port	-1	-1	-1	userconfig.properties device.server.port=

Table 3 shows configuration properties that affect the behavior of AEFSessions.

Table 3. AEFSession configuration properties

Property	Default
pos.sales.application **Note:** Set to the qualified path of the POS application to run.	SA: R::ADX_IPGM:EAMTS10L.286 ACE: R::ADX_IPGM:JSIFTS10.386 GSA: R::ADX_IPGM:EALTS10L.286
aefio.device.group **Note:** If a unique AEFIO device group is required, use the AEFIOCFG tool to create the device group, and set this property to the name of the new AEFIO device group. This step may be required if you are running Remote GUI with different remote devices.	Default
start.pos.sales.application **Note:** 1. For virtual sessions, this property controls whether the POS application is automatically started when the session is created. If a client requests a virtual session that has not yet been created, this flag determines if the POS application is started upon session creation. If set to false, the client can call AEFSession.startApplication() to cause the POS application to be launched. This is useful in situations where a client wants to observe or participate in the application startup. The client creates the session, installs any listeners and hooks, and then starts the application. 2. For real terminal sessions, the client cannot directly control whether the POS application is started, but this property still has an effect. If this property is set to false, the Toshiba applications SA, ACE, and GSA will block early in their initialization processes, waiting for the AEFSession.startApplication() to be called. Again, this is useful in situations where the client wants to observe or participate in the POS application initialization.	True
session.extensions **Note:** Allows user code to be attached to an AEFSession. Run the code in the same JVM with the session. Set to a comma delimited list of classes that will be instantiated during session initialization. If the extension objects implement the SessionExtension interface, then the object's initialize method is called.	N/A

Table 3. AEFSession configuration properties (continued)

Property	Default
application.shared **Note:** This property controls the single process load functionality. This function causes the OS to load a singleton code and constant space for the terminal sales application, thus reducing the memory footprint of the application. This property only applies to the ACE application.	ACE: True SA: False GSA: False

Note: The following overrides can be used for AEFSession configuration properties listed in Table 3 on page 18

1. To override for all sessions belonging to a session role, put the property in usersession_sessionrole.properties where sessionrole is the value defined in the usersessionrole.properties file. See the example in "Using properties file configuration" on page 175.
2. To override for a single session, put the property in usersession_*xxx*.properties where *xxx* is the terminal number.

Automation Provider

The Automation Provider is a set of Java interfaces and classes that can be accessed locally or remotely to drive POS function. The classes in the automation package implement an object oriented view of the POS application. The package contains classes that model sales transactions, operators, items, and tenders.

The method calls within the automation package are designed to be synchronous. All operations are either successful (and return a result if appropriate), or fail by throwing an exception.

The automation provider package is anchored by the POSAutomationProvider interface. Each AEFSession contains an instance of POSAutomationProvider. Operations on the POSAutomationProvider might return instances of other interfaces from the automation package. For example, calling POSAutomationProvider.startTransaction() returns an instance of SalesTransaction. The SalesTransaction interface includes addItem methods that return instances of the Item interface.

Figure 11 on page 20 illustrates retrieving the POSAutomationProvider instance from the AEFSession, and using it to execute a sales transaction.

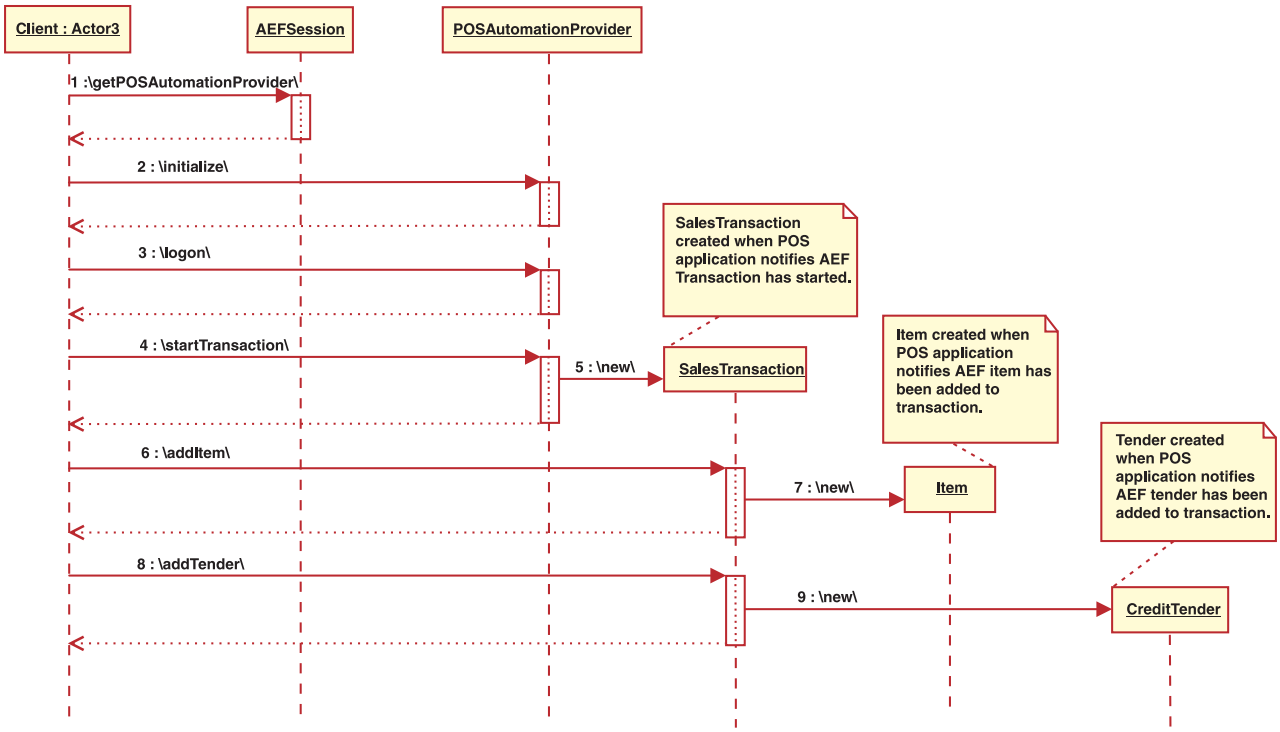

Figure 11. Execute a transaction sequence

Error handling modes

The Automation Provider drives input into the POS application, and monitors the results. When the underlying POS application experiences an error that requires operator guidance or operator intervention, or both, the result is dependent on the Automation Provider's error handling mode.

Automatic mode

In Automatic error handling mode, the AEF makes every attempt to complete the requested operation, regardless of the guidance displayed by the POS application. This includes:

- Automatically clearing informational guidance messages.
- If configured, automatically performing operator and manager overrides when required. If the POS application requires a manager number, then a valid manager override number has to be configured.
- If configured, automatically providing a legal birth date for age restricted items.
- Even with these interventions, many application errors might still occur that cannot be circumvented. In these cases, the requested operation cannot complete, and then an AEFException is thrown back to the caller.

  Automatic mode is designed for client applications where no operator such as a store clerk or cashier is present. This level of error handling is appropriate for end users who cannot be expected to deal with cashier level guidance. An example would be Web-based home shopping.

Default mode

In Default error handling mode, the AEF waits for operator intervention whenever the POS application displays operator guidance messages. The caller is blocked during the time that the AEF is waiting for the operator to resolve the guidance.

Note: If the default error handling mode is used, the RMI read timeout values specified in the config bundle have to be set to **zero** so that there is no socket read timeout while waiting on the operator. In addition, all the specific read time-outs must be set to **zero**.

Callback mode

Callback is an error handling approach, which can be used with ACE and SA, that allows the client application to receive an event when the POS application makes a request for data. The **CallbackRequest** indicates the data field or fields required. The client application is then responsible for providing any data needed to satisfy the request.

For example, if a prompt comes up requesting zip code, the callback request would be:

```
<CallbackRequest>
 <ID>125</ID>
 <RequestType>CUSTOMER_ZIPCODE_PROMPT</RequestType>
 <Prompt>ENTER ZIPCODE</Prompt>
 <DataFieldList>
 <DataField>
  <FieldID>0</FieldID>
  <Required>true</Required>
  <DataType>NUMERIC</DataType>
  <MinLength>5</MinLength>
  <MaxLength>5</MaxLength>
 </DataField>
 </DataFieldList>
</CallbackRequest>
```

The following is a sample callback response:

```
<CallbackResponse>
<CallbackResult>
 <ID>125</ID>
 <CallbackData>
  <FieldID>0</FieldID>
  <FieldData>27613</FieldData>
 </CallbackData>
 </CallbackResult>
</CallbackResponse>
```

Security:
Sometimes sensitive data can be contained in a callback response. The value in the element can be masked in the POSBC and AEF logs for extra security. The element is used in the XML message to indicate if security is enabled. It is an optional field that can be true or false. When security is enabled, the value in the element in the POSBC log will be masked to DATA_IS_SECURE. For example, enabling secure on the callback response above would look like this:

```
<CallbackResponse>
<CallbackResult>
 <ID>125</ID>
 <CallbackData>
  <FieldID>0</FieldID>
  <Secure>true</Secure>
  <FieldData>DATA_IS_SECURE</FieldData>
 </CallbackData>
 </CallbackResult>
</CallbackResponse>
```

Setting the error handling mode

The following methods of setting the error handling mode provide varying degrees of granularity, and are listed from general to most specific:

- The overall default error handling mode property setting (error.handling.mode in the automation property bundle)
- Error handling overridden per error (in *apperror.properties)
- Callback requested per request XML (for POSBC) and set per request through Identifier.setErrorHandlingMode (for AEF)

Overall error handling

The overall error handling mode property setting (error.handling.mode in the automation property bundle) can be set statically or dynamically:

Statically:
To set the error handling mode statically, set the *automation* bundle property as follows:

```
com\\ibm\\retail\\AEF\\automation\\error.handling.mode=default
```

Refer to Appendix D, "AEF property files," on page 203.

Dynamically:
To set the error handling mode dynamically when using AEF, use the POSAutomationProvider.setProperty method. The following code sets the automation provider error handling mode to default:

```
automationProvider.setProperty(POSAutomationProvider.ERROR_HANDLING_MODE,
POSAutomationProvider.HANDLE_DEFAULT);
```

Figure 12. Automation provider error handling code

Examples of error handling settings

The following examples show how the error handling settings work together and override each other:

1. For an ACE POSBC client, set your overall handling mode to *default*:

   In userappautomation.properties, add:
   `com\\ibm\\retail\\AEF\\automation\\error.handling.mode=default.`
2. To change the error handling to automatic for all ENTER PRICE OR PRESS CLEAR prompts, in your userapperror.properties, add:

   `11033_ERROR_HANDLING_MODE=1`
   where 1 is automatic, 2 is default, 3 is callback, and -1 is not specified.
3. Then, to set any errors during add item to callback mode (overriding the "automatic" handling mode that was just set for ENTER PRICE OR PRESS CLEAR prompts), use the following XML where callback is set to true:

```
<xsd2:AddItem xmlns:xsd2="http://bc.si.retail.ibm.com/POSBCSchema">
 <AddItemRequest>
  <RequestID>2</RequestID>
  <ItemIdentifier>
   <KeyedItemID>1</KeyedItemID>
  </ItemIdentifier>
  <EnableCallback>true</EnableCallback>
 </AddItemRequest>
</xsd2:AddItem>
```

Identifier arguments

Many of the interfaces within the automation package have pairs of methods that differ in the complexity of their arguments. For example, within the POSAutomationProvider interface, a logon method takes an explicit operator id, and then an explicit password:

public Operator logon(String operatorID, String password) throws RemoteException, AEFException;

The code to logon using this method is:

```
// Logon with operator id = 542, password = 335.
Operator operator = automation.logon("542","335");
```

An equivalent logon method takes an instance of OperatorIdentifer.

public Operator logon (OperatorID operatorID) throws RemoteException, AEFException;

The OperatorIdentifer interface contains methods to set the operator ID and password.

Each (obj_type)Identifier interface is implemented by the (obj_type)IdentiferImpl class. Following the example above, an instance of OperatorIdentifierImpl could be used as an argument to the logon method. Each of the IdentifierImpl classes includes constructor signatures that include the commonly required arguments.

The following code shows the use of an OperatorIdentifier in the logon method.

```
// Create an operator id with id = 542, password = 335.
OperatorIdentifier opID = new OperatorIdentifer("542", "335");
Operator operator = automation.logon(opID);
```

Two more important points about Identifer objects as arguments are as follows:

1. Identifier objects are useful as parameters because they derive from the Hashmap class and therefore allow any key/value pair. This provides for API extension when additional arguments must be included because of application customization.
2. Identifier interfaces have an inheritance structure. If an automation method is defined to take an instance of an Identifier interface, it is valid to pass an instance of an object implementing a subclass of that Identifier interface. In some cases, it is necessary to pass a subclass. For example, the SalesTransaction.addTender method takes a TenderIdentifier instance. However, a subclass of this interface, such as CreditIdentifier or MSRCreditIdentifier, should be passed to determine the type of tender.

   Figure 13 illustrates the hierarchy of Identifer interfaces.

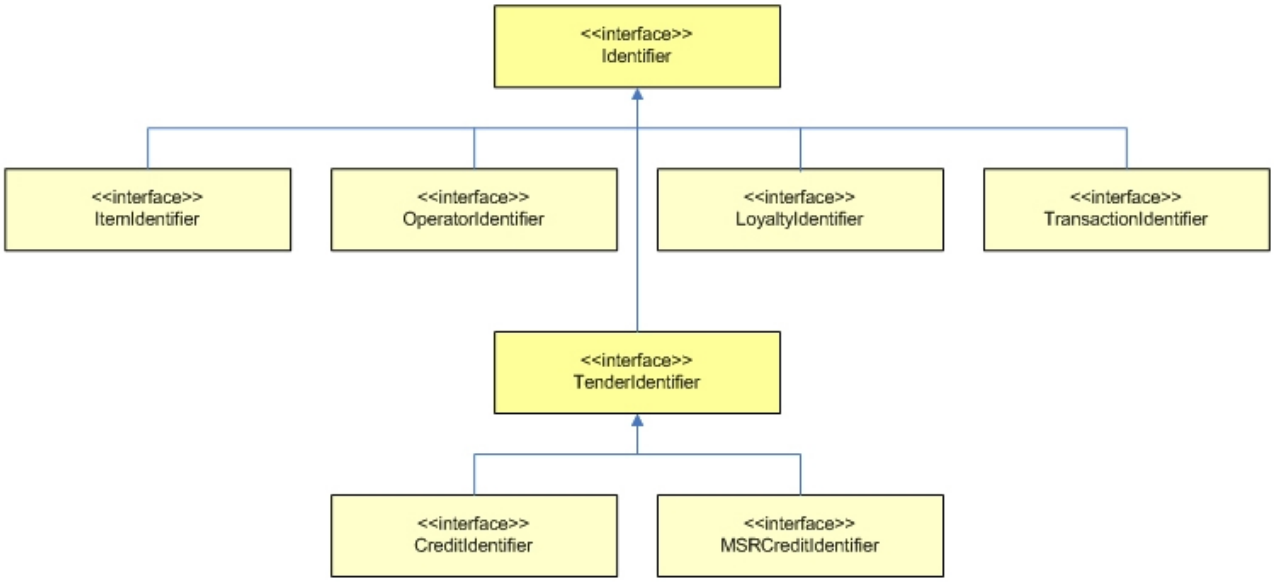

Figure 13. Identifier interface hierarchy

Info objects

The interfaces within the automation package provide for remote access. When an object obtains a reference to a remote object, all the method calls to that object require a round trip on the network. To optimize performance, a remote client should minimize the number of remote method calls. To facilitate this, AEF architecture encapsulates much of the remote object's instance data into a serialized info object.

For example, when an Operator instance is returned from a POSAutomationProvider.logon call, the Operator.getInfo method can be called to return an instance of OperatorInfo. The OperatorInfo object contains method to retrieve the operator name, ID, and password. The info object allows all three attributes to be returned to the client in a single method call, rather than a separate remote call for each attribute.

Figure 14 shows the Info interfaces and the objects that own their instances.

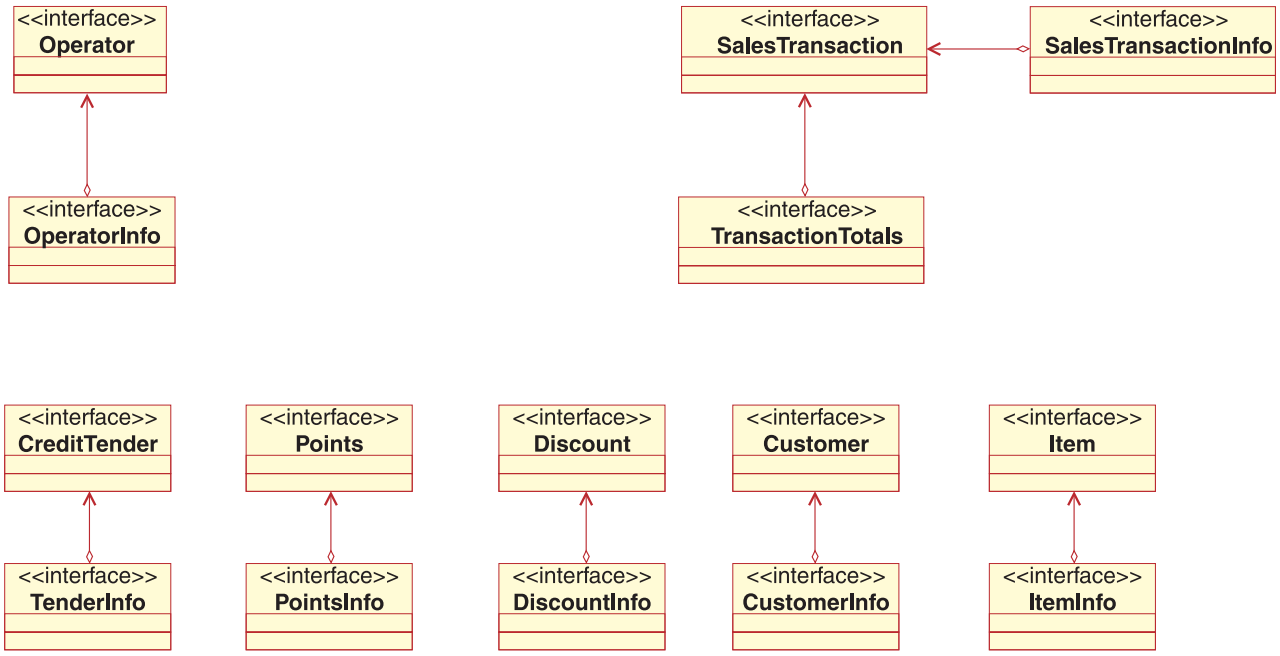

Figure 14. Info object interfaces

Configuration properties

The AEF Automation Provider can be configured using properties in the *automation* bundle. See Table 4.

Table 4. Automation Provider configuration properties

Property name	Default value	Valid values	Notes
com\\ibm\\retail\\AEF\\automation\\ automatic.manager.override	true	true, false	Controls whether or not the Automation Provider automatically performs a manager override when required.
com\\ibm\\retail\\AEF\\automation\\ automatic.manager.override.number	1 for SA; 9 for ACE	any numeric value	For SA and ACE, this is the manager override used for automatic overrides. It must be a valid override number configured in the application.

Table 4. Automation Provider configuration properties (continued)

Property name	Default value	Valid values	Notes
com\\ibm\\retail\\AEF\\automation\\ automatic.operator.override	true	true, false	For SA and ACE, controls whether the Automation Provider automatically performs an operator override when required.
com\\ibm\\retail\\AEF\\automation\\ item.age.restriction	true	true, false	For SA and ACE, controls whether Automation Provider automatically handles a prompt for birth date for age restricted items. If true, Automation Provider provides a fixed legal birth date for the prompt.
com\\ibm\\retail\\AEF\\automation\\ automatic.special.signon	true	true, false	For SA and ACE, controls whether Automation Provider automatically performs a special signon for sessions that are in secure signoff mode.
com\\ibm\\retail\\AEF\\automation\\ error.handling.mode	automatic	automatic, default, callback (for ACE)	Determines whether Automation Provider error handling logic attempts to automatically handle operator guidance (automatic mode), or wait for an operator (default), or raise an event to enable the client application to supply data requested by the POS application (callback).
com\\ibm\\retail\\AEF\\automation\\ error.max.errors.to.handle	10	numeric	Specifies the number of sequential errors the Automation error handler should attempt to handle before failing the request. This is used in situations where the application might be looping through a set of two or more errors.

Table 4. Automation Provider configuration properties (continued)

Property name	Default value	Valid values	Notes
com\\ibm\\retail\\AEF\\automation\\ automatic.till.exchange	true	true, false	For SA and ACE, determines whether the Automation Provider performs a till exchange automatically when required by the application.

POSAutomationProvider

The POSAutomationProvider interface provides such functions as operator signon and signoff, starting and voiding transactions, and so on. Each AEFSession contains an instance of a POSAutomationProvider object.

Operator

An instance of the Operator interface is returned from a successful call to the POSAutomationProvider.logon method, or by calling POSAutomationProvider.getOperator if an operator is already logged onto the application.

Each Operator instance contains an instance of OperatorInfo, and OperatorAuthorization. OperatorInfo contains the operator ID, name, and password. OperatorAuthorization contains application specific values indicating which functions the operator is authorized to perform.

Customer

An instance of the Customer interface is returned from a successful call to the SalesTransaction.addCustomerLoyaltyID method, or by calling the SalesTransaction.getCustomer method if a loyalty ID has already been added to the transaction.

Each instance of the Customer interface contains an instance of CustomerInfo. The CustomerInfo interface includes information such as customer ID, name, points totals and balances, and targeted coupons and messages.

Transaction

An instance or a subclass of the Transaction interface is returned from a successful call to the POSAutomationProvider.startTransaction method, or by calling the POSAutomationProvider.getTransactionmethod if a transaction is already in progress. The only type of transactions that are supported are sales transactions. Therefore, calls to the POSAutomationProvider.startTransaction method return the SalesTransaction sub-interface of the Transaction interface.

Each Transaction instance contains an instance of TransactionInfo. TransactionInfo contains the transaction ID (transaction number), transaction date, and a void indicator.

SalesTransaction

The SalesTransaction interface represents the only type of transaction supported in this release of SI. The SalesTransaction interface supports adding, removing, and returning items, adding and removing tenders, adding customer loyalty numbers, and suspending the transaction.

Each SalesTransaction instance contains an instance of SalesTransactionInfo and TransactionTotals. SalesTransactionInfo includes a tax exempt indicator. TransactionTotals includes the transaction amount, balance due, and so on. TransactionTotals objects are current only at the time they are retrieved. They include the following transaction amounts: total, subtotal, tax, balance due, foodstamp balance due, foodstamp total, change due, foodstamp change due, coupon total, total items, total coupons, and loyalty savings.

LineItem

The LineItem interface represents a single item within the transaction. Line items include item sold, coupons, tenders, discounts, and loyalty points. There are sub-interfaces of the LineItem interface for each of the line item types.

Some of the SalesTransaction methods such as addItem return an ArrayList of LineItems. The client can iterate through the array and use the instance of LineItemInfo to determine each line item's type. For example, because of linked items and electronic coupons, a single addItem call may result in an ArrayList containing multiple Item elements, as well as Coupon elements. Note that both Item and Coupon are extensions of the LineItem interface type.

The LineItem interface includes support for voiding the line item, fetching the LineItemInfo object, and fetching the item's identifier.

The LineItemInfo includes a description, receipt lines, a void indicator, refund indicator, and deposit indicator. Usually, the instance of LineItem is an instance of a sub-interface of LineItem (such as Coupon, Item, and so on). Therefore, the LineItemInfo is an instance of a sub-interface such as CouponInfo or ItemInfo.

Item

The Item interface represents a purchase item within the transaction. The Item.getInfo method returns an instance of ItemInfo. Each Item contains an instance of ItemInfo.

The ItemInfo interface includes item identifier, item pricing information, quantity, weight, age restriction, foodstamp indicator, WIC indicator, restricted sales periods, item repeat indicator, and tax information.

Coupon

The Coupon interface represents a store or manufacturer coupon within the transaction. The Coupon.getInfo method returns an instance of CouponInfo. Each Coupon contains an instance of CouponInfo. The CouponInfo interface includes a coupon type, as well as sharing many of the Item attributes.

Discount

The Discount interface represents a line item or transaction discount within the transaction. The Discount.getInfo method returns an instance of DiscountInfo. Each Discount contains an instance of DiscountInfo. The DiscountInfo interface includes information about the type of discount, the discount rate or amount, the discount reason, a taxability indicator, and whether the discount applies to the entire transaction or a specific item.

Points

The Points interface represents the awarding or redemption of customer loyalty points within a transaction. Points objects are LineItem instances and can be automatically generated by the POS application in response to selling an item. Each Points object contains an instance of PointsInfo. The PointsInfo interface includes information such as the type of points, whether the points are awarded or redeemed, and the number of points involved.

Tender

The Tender interface represents a tender LineItem within the transaction. The tenders supported for the SalesTransaction.addTender method are cash, credit, debit, gift card, EBT cash, and EBT food stamp tenders. An instance of CashIdentifierImpl, CreditIdentifierImpl, DebitIdentifierImpl, GiftCardIdentifierImpl, EBTCashIdentifierImpl, EBTFoodStampIdentifierImpl, or MSRCreditIdentifierImpl should be passed to the SalesTransaction.addTender method. The TenderInfo interface includes the tender amount.

Using remote references

The Automation Provider objects listed above are remote objects. If a remote client holds a reference to an object instance, garbage collection for this object cannot be performed in the host JVM. If a client application is ill behaved or exhibits a memory leak, the host JVM can experience a memory leak because garbage collection cannot be performed for the remote objects referenced by the client until the client de-references the objects.

An example of this is a remote client program that attaches to a real terminal session. The remote client gets a reference to the SalesTransaction method. The operator on the real register then ends the transaction. There are no more references to the SalesTransaction instances on the register, but garbage collection for this object cannot be performed until the client de-references the remote reference by nulling the SalesTransaction reference, letting the reference go out of scope, terminating the application, or losing network connectivity to the host JVM.

Automation API

This section describes how the Automation Provider API is implemented. An understanding of the Automation Provider is required to customize its behavior. Modifications or extensions to the AEF might be necessary to AEF enable any user modifications to the POS application.

Action classes

The subset of Automation Provider methods that drive input into the POS applications is implemented using action classes. The Automation Provider methods that make use of action classes include the following line in the class JavaDoc:

`Action ID = ActionKey`

You can look up the *ActionKey* in the **class** bundle. See “Class Bundle” on page 203 to determine the class used to implement the action. Table 5 lists the AEF action classes.

Table 5. Base action classes

AEFActionImpl.java
AEFCompoundActionImpl.java
FuelSaleToPipeActionImpl.java
InitializeActionImpl.java
PreAuthToPipeActionImpl.java
ScanDataActionImpl.java
SwipeMSRActionImpl.java
UserTLogDataToPipeActionImpl.java
UserTLogExceptionToPipeActionImpl.java

Table 6. ACE action classes

ACEActionImpl.java
ACEApplyDelayedCouponsActionImpl.java
ACECancelOverrideActionImpl.java
ACECancelPreviousEntryActionImpl.java
ACECashTenderActionImpl.java
ACECustomerEntryActionImpl.java
ACECouponActionImpl.java
ACEDataOnPipeActionImpl.java
ACEEBTFoodStampTenderActionImpl.java
ACEEPSEFTTenderActionImpl.java
ACEFoodStampTenderActionImpl.java
ACEForcedLogoffActionImpl.java
ACEItemEntryActionImpl.java
ACEItemReturnActionImpl.java
ACEItemVoidActionImpl.java
ACELogoffActionImpl.java
ACELogonActionImpl.java
ACEManagerOverrideActionImpl.java
ACEMiscTenderActionImpl.java
ACENoSalePriceVerifyActionImpl.java
ACEOperatorOverrideActionImpl.java
ACEPreAuthRequestActionImpl.java
ACEPrintGiftReceiptActionImpl.java
ACERequestSuspendedFuelActionImpl.java
ACERetrieveSuspendedTransactionListActionImpl.java
ACERetrieveTransactionActionImpl.java
ACESetTrainingModeOnActionImpl.java
ACEStartTransactionActionImpl.java
ACESuspendTransactionActionImpl.java
ACEVoidTransactionActionImpl.java
ACETransactionDataOnPipeActionImpl.java
ACEVoidCouponActionImpl.java

Table 7. GSA action classes

GSAActionImpl.java
GSACancelPreviousEntryActionImpl.java
GSACashTenderActionImpl.java
GSACheckTenderActionImpl.java
GSAClearActionImpl.java
GSACouponActionImpl.java
GSAEFTCreditTenderActionImpl.java

Table 7. GSA action classes (continued)

GSAForcedLogoffActionImpl.java
GSAGenericTenderActionImpl.java
GSAItemEntryActionImpl.java
GSAItemReturnActionImpl.java
GSAItemVoidActionImpl.java
GSALogoffActionImpl.java
GSALogonActionImpl.java
GSAManagerOverrideActionImpl.java
GSARetrieveSuspendedTransactionListActionImpl.java
GSARetrieveTransactionActionImpl.java
GSASetTrainingModeOnActionImpl.java
GSAStartTransactionActionImpl.java
GSASuspendTransactionActionImpl.java
GSATotalActionImpl.java
GSAVoidCouponActionImpl.java
GSAVoidTenderActionImpl.java
GSAVoidTransactionActionImpl.java

Table 8. SA action classes

SAActionImpl.java
SAApplyDelayedCouponsActionImpl.java
SACashTenderActionImpl.java
SACancelOverrideActionImpl.java
SACancelPreviousEntryActionImpl.java
SAClearActionImpl.java
SACustomerEntryActionImpl.java
SAEFTCreditTenderActionImpl.java
SAEFTTenderActionImpl.java
SAEnterStandAloneActionImpl.java
SAForcedLogoffActionImpl.java
SAItemEntryActionImpl.java
SAItemReturnActionImpl.java
SAItemVoidActionImpl.java
SALogoffActionImpl.java
SALogonActionImpl.java
SAManagerOverrideActionImpl.java
SAOperatorOverrideActionImpl.java
SAPriceVerifyActionImpl.java
SARetrieveSuspendedTransactionListActionImpl.java
SARetrieveTransactionActionImpl.java
SASetTrainingModeOnActionImpl.java

Table 8. SA action classes (continued)

SAStartTransactionActionImpl.java
SASuspendTransactionActionImpl.java
SAVoidCouponActionImpl.java
SAVoidTransactionActionImpl.java

When an action is constructed, it is passed an instance of ActionRequest that contains a HashMap of all the arguments necessary to perform the action. The action performs its function when its performAction method is called. In general, the steps taken in the performAction method are as follows:

- Check to see if the POS application is currently in an error state that must be handled prior to performing the action.
- Check any preconditions that must hold prior to performing the action. Examples are actions that require the POS application to be in a specific state or substate to perform the action.
- Create a SimpleKeySequenceAction based on arguments from the ActionRequest. The SimpleKeySequenceActionImpl sends a key sequence to the POS application.
- Define the expected results of the key sequence. This includes good and bad conditions. Good conditions result when the POS application reacts to the key sequence as expected. Bad conditions result when the POS application reacts to the key sequence by generating an error requiring operator guidance, or other results that require AEF error handling.
- Use a ConditionLock to perform the SimpleKeySequenceActionImpl and to wait for one of the good or bad conditions to be observed. If a bad condition is detected, pass control to the AEF error handler to resolve.
- If the action is supposed to return an object or objects (such as an Operator or LineItems), then it uses an (ObjectDetectorLock) to wait until the objects are detected.

  See Appendix G, “SALogonActionImpl source example,” on page 273 for a detailed discussion of the SALogonActionImpl class.

Property conditions

A property condition is a logical condition predicated on a POSDataProvider property. Property conditions evaluate to true or false depending on the value of a POSDataProvider property. Property condition classes check for value equality such as: value greater than, value less than, value not equal to, and so on.

The action classes discussed above use property conditions to define good and bad results when injecting input into the POS application. For example, we may want to define a good condition that the POS application is in state 10 and is either in substate 1001 or 1008. Since property conditions are always defined as arrays in Java, a good condition can be represented with the following declaration:

```
Condition substateConditions[] = {
         new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
               POSDeviceProperties.POS_SUB_STATE, "1001"),
         new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
               POSDeviceProperties.POS_SUB_STATE, "1008") };

   Condition andConditions[] = {
         new OrCondition(substatesConditions),
         new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
               POSDeviceProperties.POS_STATE, "10") };

   Condition goodConditions[] = { new AndCondition(andConditions) };
```

The condition array goodConditions can be passed to a ConditionLock to define an expected good result of injecting POS input.

All conditions within the condition array should be logically compared together using the logical **or** operand. In the example above, the final array goodConditions has only a single element. The following example declares a condition array that is true if any of the elements are true because the array elements are logically compared using **or** in the command string:

```
Condition badConditions[] =
{
   // All 4 conditions are "or'ed" so this set of conditions
   // is true if any of the substates (10333, 10336, 10339,
   // 10343) are observed.
   new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
                               POSDeviceProperties.POS_SUB_STATE,
                               "10333"),
   new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
                               POSDeviceProperties.POS_SUB_STATE,
                               "10336"),
   new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
                               POSDeviceProperties.POS_SUB_STATE,
                               "10339"),
   new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
                               POSDeviceProperties.POS_SUB_STATE,
                               "10343"),
}
```

Figure 15. Condition array example

The AEF includes Java classes that support the property value comparisons shown in Table 9.

Table 9. Property condition classes

Java class name **com.ibm.retail.AEF.automation package**	**Operation**	**Example**

PropertyEqualsCondition	Checks a property value for equality with the specified value. Specified value may be a string or a number.	A property condition that determines when the POSDataProvider property (property category = POS_DEVICE, property name = subState) equals "1008" could be defined as:
		`PropertyEqualsCondition("POS_DEVICE", "subState", "1008")`
		
		Another way to specify this same condition is:
		`PropertyEqualsCondition(POSDeviceProperties .CATEGORY, POSDeviceProperties.POS_SUB_STATE, Substate.getSubstate("SELECT_PROCEDURE"))`

PropertyNotEqualsCondition	Checks a property for non-equality with the specified value. Specified value may be a string or a number.	A property condition that determines when the current state is not the clear state (state 1) could be defined as: `PropertyNotEqualsCondition(POSDeviceProperties. CATEGORY, POSDeviceProperties.POS_STATE, State. getState("CLEAR"))`
PropertyGreaterThanCondition	Checks a property value to see if it is greater than the specified value. The property value must be able to be converted to a numeric value.	A property condition that indicates that the number of customer loyalty points is greater than 5000 could be defined as: `PropertyGreaterThanCondition(PointsProperties. CATEGORY, PointsProperties.TOTAL, 5000)`
PropertyGreaterOrEqualCondition	Checks a property value to see if it is greater than or equal to the specified value. The property value must be able to be converted to a numeric value.	A property condition that indicates when the transaction total is greater than or equal to zero could be defined as: `PropertyGreaterorEqualCondition(TransactionTotals Properties.CATEGORY, TransactionTotalProperties. TOTAL, 0)`
PropertyLessThanCondition	Checks a property value to see if it is less than the specified value. The property value must be able to be converted to a numeric value.	A property condition that indicates when the scale weight drops below 1 could be defined as: `PropertyLessThanCondition(POSDeviceProperties. CATEGORY, POSDeviceProperties.SCALE_WEIGHT_VALUE, 1)`
PropertyLessOrEqualCondition	Checks a property value to see if it is less than or equal to the specified value. The property value must be able to be converted to a numeric value.	A property condition that indicates when the transaction total is less than or equal to 5.25 could be defined as: `PropertyLessOrEqualCondition(TransactionTotals Properties.CATEGORY, TransactionTotalProperties. TOTAL, 5.25)`

Table 9. Property condition classes (continued)

Java class name com.ibm.retail.AEF.automation package	Operation	Example
PropertyContainsCondition	Check a property value to see if it contains the specified substring.	A property condition that indicates when the first line of the line display contains the string ERROR could be defined as: `PropertyContainsCondition(POSDeviceProperties. CATEGORY, POSDeviceProperties.ANPROMPT_LINE1, "ERROR")`
PropertyNotContainsCondition	Indicates when a property value does not contain the specified substring.	A property condition that indicates when the first line of the customer display does not contain the string "WELCOME" could be defined as: `PropertyContainsCondition(POSDeviceProperties. CATEGORY,POSDeviceProperties.CUST_PROMPT_LINE1, "WELCOME")`
PropertyContainsAtIndexCondition	Indicates when a property contains the specified substring at the specified (0 based) index.	A property condition that indicates when the first line of the customer display contains the substring "CREDIT" starting at the third character of the line could be defined as: `PropertyContainsAtIndexCondition(POSDevice Properties.CATEGORY, POSDeviceProperties.CUST_PROMPT_LINE1, "CREDIT", 2)`
PropertyNotContainsAtIndex Condition	Indicates when a property does not contain the specified substring at the specified (0 based) index.	A property condition that indicates when a customer's loyalty number expiration date does not contain "04" in the 4th position could be defined as: `PropertyNotContainsAtIndexCondition(Customer Properties.CATEGORY, CustomerProperties.ID_EXPIRATION_DATE, "04", 3)`
PropertyRegexMatchCondition	Indicates when a property value matches the specified regular expression.	A property condition that indicates when the first line of the customer display starts with the letter **B** followed by three digits could be defined as: `PropertyRegexMatchCondition(POSDeviceProperties. CATEGORY, POSDeviceProperties.CUST_PROMPT_ LINE1, "B\\d\\d\\d")`
PropertyRegexNotMatchCondition	Indicates when a property value does not match the specified regular expression.	A property condition that indicates when the last item scanned does not begin with a letter could be defined as: `PropertyRegexNotMatchCondition(POSDeviceProperties. CATEGORY, POSDeviceProperties.LAST_SCAN_ LABEL, "[a-zA-z]")`
AndCondition	Evaluates true when all the child conditions evaluate true.	If badConditions is an array holding property conditions, a new AndCondition containing the badConditions array as children could be declared as: `AndCondition(badConditions)`
OrCondition	Evaluates true when any of the child conditions evaluate true.	If goodConditions is an array holding property conditions, a new OrCondition containing the goodConditions array as children could be declared as: OrCondition(goodConditions) **Note:** Passing an array of Condition objects to a ConditionLock causes the array element conditions *to be declared using the logical* or, so it might not be necessary to create an explicit OrCondition.
AndThenCondition	Similar to the AndCondition, except that initially, only the first child condition is eligible for evaluation. If the first child condition evaluates to true, then the second child condition becomes eligible for evaluation, and so on. When all children conditions evaluate to true, the overall AndThenCondition evaluates to true.	If badConditions is an array holding property conditions, a new AndThenCondition containing the badConditions array as children could be declared as: `AndThenCondition(badConditions)`

Bad conditions

Since a common set of bad conditions is often used repeatedly by the actions, they have been made available using the following method calls:

```
// Standard bad conditions
BadConditionsImpl.getInstance().getBadConditions();
// Bad conditions for EFT/EPS
BadConditionsImpl.getInstance().getExtendedBadConditions();
```

The set of bad conditions returned from each method is dependent on the POS application, and can be viewed in the following source modules:

```
ACEBadConditionsImpl.java
GSABadConditionsImpl.java
SABadConditionsImpl.java
```

ConditionLock

The ConditionLock is used to inject keyboard or MSR input into the POS application, and to wait for one of a set of specified property conditions to be observed. The conditions are usually defined as a set of good conditions, and a set of bad conditions.

The following code fragment shows a call to a ConditionLock to cause the key sequence to be injected. The return code indicates which of the good or bad conditions was observed.

```
HashMap args = new HashMap();
  args.put("%0", password);
  args.put("SEQUENCE_ID", "password");
  ConditionLock lock = new ConditionLock();
  AEFAction keySequenceAction = (AEFAction) (actionFactory.makeAction(
        new ActionRequest("SimpleKeySequenceAction", args)));
  retVal= lock.performActionAndWait("wait-for-item-entry-state",
        keySequenceAction, goodConditions,
        BadConditionsImpl.getInstance().getBadConditions(), getTimeout());
```

In the previous example, a SimpleKeySequenceAction is created that sends the key sequence to enter an operator password. The ConditionLock is used to perform the action. The first argument of the performActionAndWait method is a string used for trace that is logged if the call times out. The key sequence is passed, along with the good and bad conditions. Finally, a timeout value is passed. Use the default timeout that is configured in the config bundle “Configuration Bundle” on page 204. The performActionAndWait call either returns with an exception or an integer value. The call returns with an exception if a timeout is waiting for any of the conditions to be observed, or if there is some other error. If one of the conditions is observed, an integer is returned. This integer indicates which condition evaluated to true. An index of zero or greater indicates one of the good conditions. The first element in the good condition array is 0, the second is index 1, and so on. A negative return value indicates one of the bad conditions was observed. The first element in the bad condition array is index -1, the second is index -2, and so on. If a bad condition is observed, the action class usually passes control to the AEFErrorHandler to attempt to resolve the error.

ObjectDetectorLock

The ObjectDetectorLock is used in conjunction with an object detector to block the calling thread until a new instance of a specific object type is detected. The waitForNewObject method blocks the calling thread until a new object is detected, or until a timeout occurs. The waitForNewObjectOrError method adds the capability to unblock if one of the specified error conditions is observed. The automation action classes such as the SALogonActionImpl class use instances of the ObjectDetectorLocks to wait for the item to be returned, such as a new Operator instance. The object detectors themselves may be retrieved from the AEFSession instance.

AEF error handling

When an error condition (as defined by a set of bad conditions) is detected as a return value from a ConditionLock within an action class, the action usually creates an instance of the AEFErrorHandler class. When created, the AEFErrorHandler attempts to determine which error helper should be created to process the error.

The error helper's job is to either resolve the error, or throw an exception if the error cannot be resolved.

The error helper class to use is defined in the **error** bundle as described in "Using the Error Bundle" on page 200. For a particular error key, there is a set of entries in the error bundle. One of the entries is the error helper class key. An error code and extended error code is also defined for each error key. These are the error codes that are used if the error helper cannot resolve the error and must throw an exception.

The following lines from the SA version of apperror.properties correspond to the B069 error key.

```
# B069 Tenders/Coupons Must be Returned
B069_ADDITIONAL_TEXT=A transaction has been voided after coupons
or other tenders have been accepted.
B069_HANDLING_CLASS_KEY=SAErrorHelperB069
B069_ERROR_CODE=PROCEDURE_NOT_ALLOWED
B069_EXTENDED_ERROR_CODE=NONE
```

The error helper class key that corresponds to an error key of B069 is SAErrorHelperB069. The error helper key SAErrorHelperB069 is looked up in the classes bundle to determine the error helper class. In this case, the error helper class is com.ibm.retail.AEF.util.SAErrorHelperB069.

The SAErrorHelperB069 will clear the error if the AEF error handling mode is automatic. If the error handling mode is default, the error helper will wait for the operator to clear the error.

Determining the error key

When an AEFErrorHandler is created, it must determine the error key that is used to look up the error helper class key as described above. Depending on the POS application, the error key is usually based on the application state or substate, or an **A***nnn* or **B***nnn* guidance number on the operator display. The algorithms for determining the error key are described below.

ACE error key algorithm

When an AEFErrorHelper is created for ACE, the current ACE substate is used to lookup the error key in appsubstates.properties (see "appclasses.properties" on page 203). For example, if the current ACE substate is 10069, then the error key is B069 as specified by the following lines in appsubstates.properties:

```
# B069 TENDERS/COUPONS MUST BE RETURNED
10069=B069
```

If the error key does not exist in appsubstates.properties or in apperror.properties, then the error key is set to APP_DEFAULT.

GSA error key algorithm

When an AEFErrorHelper is created for GSA, if the operator display contains an **A***nnn* guidance number, then the guidance number is used as the error key. If this error key is not listed in apperror.properties, then the entire first line of the operator display is used to lookup a descriptor number. For example, if the first line of the operator display matches the descriptor number 35 as defined in GSA, then the error key is BASE_DESC35. If this error key is not listed in apperror.properties, then the current application state is appended onto STATE_ to form the error key. For example, if the current GSA state is 20, then the error key is STATE_20. If this error key is not listed in apperror.properties, then the error key is set to APP_DEFAULT.

SA error key algorithm

When an AEFErrorHelper is created for SA, if the operator display contains a **B***nnn* guidance number, then the guidance number is used as the error key. If this error key is not listed in apperror.properties, then the entire first line of the operator display is used to lookup a descriptor number. For example, if the first line of the operator display matches the EFT descriptor number 20 as defined in SA, then the error key is EFT_DESC20. If this error key is not listed in apperror.properties, then the current application substate is used as the error key. If this error key is not listed in apperror.properties, then the error key is set to APP_DEFAULT.

Error helper classes

Table 10 lists the error helper classes available in the AEF. Instructions for writing a new error helper classes are included in Chapter 3, "Programming with the AEF," on page 93.

Table 10. Error helper classes

Error helper key	Error helper behavior	Implementation classes available for applications
ACEErrorHelperB004 SAErrorHelperB004	Used to handle the B004 MANAGERS KEY IS REQUIRED prompt. If the error handling mode is set to *automatic*, and the Automation Provider is configured for automatic manager override, the error helper clears the error, and sets the keylock position to manager mode. If the error handling mode is default, the error helper waits for the operator to clear the error, and then an exception is thrown.	ACE, SA
ACEErrorHelperB005 SAErrorHelperB005	Used to handle the B005 REMOVE MANAGERS KEY prompt. If the error handling mode is set to automatic, and the Automation Provider is configured for automatic manager override, the error helper clears the error, and restores the keylock position if necessary. If the error handling mode is default, the error helper waits for the operator to clear the error, then an exception is thrown.	ACE, SA
ACEErrorHelperB013 SAErrorHelperB013	Used to handle the B013 CLOSE THE CASH DRAWER prompt. In both automatic and default error handling modes, the error helper waits for the cash drawer to be closed. If the cash drawer is emulated, it will close automatically. If the cash drawer is not emulated, the operator must close it.	ACE, SA
ACEErrorHelperB040 SAErrorHelperB040	Used to handle the B040 FILE ACCESS FAILED message. If the error handling mode is set to automatic, the error helper clears the error, performs a manager override if the error is eligible to be overridden, and the Automation Provider is configured for automatic override. Finally, if the application prompts to enter standalone, the error helper performs the key sequence to enter standalone mode. If the error handling mode is default, the error helper waits for the operator to clear the error. The error helper then waits for the operator to perform or cancel a manager override if the override prompt appears. Finally, the error helper waits for the operator to perform the key sequence to enter standalone if prompted to do so. In either error handling mode, an exception is thrown.	ACE, SA
ACEErrorHelperB069 SAErrorHelperB069	Used to handle the B069 TENDERS/COUPONS MUST BE RETURNED message. If the error handling mode is set to automatic, the error handler will clear the message and, for a suspend, reissue the suspend sequence if only coupons were outstanding. Otherwise, the error is fatal.	ACE, SA
ACEErrorHelperB085 SAErrorHelperB085	Used to handle the B085 TILL EXCHANGE REQUIRED message. If the error handling mode is set to automatic and the Automation Provider is configured to perform automatic till exchanges, then the error helper clears the error, and sends the till exchange key sequence. If the error handling mode is default, the error helper clears the error, and throws an exception.	ACE, SA
SAErrorHelperB113	Used to handle the B113 REMOVED FRANKED TENDER message. For both automatic and default error handling modes, the error helper waits for the document to be ejected from the printer.	SA
SAErrorHelperB537	Used to handle the B537 NO STAND-IN FOR THIS SERVICER message. If the error handling mode is automatic, the error helper clears the error and throws an exception. If the error handling mode is default, the error helper waits for the operator to clear the error, then throws an exception.	SA

Table 10. Error helper classes (continued)

Error helper key	Error helper behavior	Implementation classes available for applications
SAErrorHelperB608	Used to handle the B608 ASK FOR CARD CLEAR TO CONTINUE prompt. There are two reasons for getting this message. One is a reminder to get the customer's card when the total key is pressed. The other is when a coupon is entered that is limited to preferred customers. If the error handling mode is automatic, the error helper clears the error. If the prompt was generated in response to a coupon that is limited to preferred customers, the error helper throws an exception. If the error handling mode is default, the error helper waits for the operator to clear the error. If the prompt was generated in response to a coupon that is limited to preferred customers, the error helper throws an exception.	SA
ACEErrorHelperB676 SAErrorHelperB676	Used to handle the B676 ACTIVE SUSPENDS, CLEAR OR CONTINUE prompt. If the error handling mode is automatic, the error helper clears the error to allow the signoff. If the error handling mode is default, the error helper waits for the operator to clear the error to allow the signoff.	ACE, SA
AcceptOrVoidTender	Used to handle the ENTER TO ACCEPT OR CLEAR TO VOID tender prompt in GSA. If the error handler is in automatic mode, the error helper sends the ENTER key to accept the tender. If the error handler is in default mode, the error helper waits for the operator to handle the prompt. If the operator clears the tender, an exception is thrown.	GSA
AccountNumberErrorHelper	GSA: Used to handle GSA state 51. In automatic mode, the error helper retrieves the account number from the tender identifier and sends it. If none is found, an exception is thrown. In manual mode, the AEF waits for the operator to enter an account number. SA: Used to handle B048 ACCOUNT NUMBER NEEDED guidance message. In automatic mode, this is a fatal error, the guidance is cleared, and an exception is thrown. In manual mode, the AEF waits for the operator to enter an account number.	GSA, SA
AgeRestrictedHelper	ACE: Used to handle date of birth prompt for age restricted items. GSA: Used to handle A399 AGE RESTRICTED ITEM guidance message. SA: Used to handle B107 AGE CHECK NEEDED guidance. In automatic mode, if the Automation Provider is configured to handle age restricted items automatically, a hard coded date of birth is entered to satisfy the age prompt. If the Automation Provider is not configured to automatically handle age restricted items, or the error handling mode is default an exception is thrown.	ACE, GSA, SA
AppDefaultErrorHelper	This is the error helper used whenever an error key cannot be located in apperror.properties. An exception is always thrown. This usually indicates that an error helper should be configured.	ACE, GSA, SA
AuthorizationCodeErrorHelper	Used to handle the ENTER AUTHORIZATION CODE prompt. In either automatic or default error handling modes, the error helper uses the authorization code from the tender identifier as input for a key sequence to satisfy the prompt.	ACE
BaseDescriptor449ErrorHelper	This error helper might be used when the error handler encounters a blank line on the operator display. The error helper waits for the line display to change.	GSA

Table 10. Error helper classes (continued)

Error helper key	Error helper behavior	Implementation classes available for applications
CardValidationNumber ErrorHelper	Used to handle the ENTER CARD VALIDATION NUMBER prompt. For automatic or default error handling mode, the error helper extracts the card validation number from the TenderIdentifier. If none is provided, an exception is thrown.	ACE
CheckId Error Helper	Handles the prompts for check tenders requesting check id information. For automatic mode, the error helper will extract the data needed from the TenderIdentifier. If none is provided, an exception is thrown.	ACE
ConfirmPrice Error Helper	Handles a situation that requires a price confirm. For automatic mode it will send an ENTER. For default mode, a new price can be entered if the price that is trying to be confirmed is incorrect.	ACE
DateOfBirth Error Helper	This error helper handles age restricted items and check Ids in ACE, specifically the request for a customer's birth date. For automatic error handling mode, this implementation will send a default birth date (01/01/1960) for age restricted items. If the prompt is for the date of birth for a check Id, the date of birth in the check Id object will be used.	ACE
DuplicateReceiptHelper	Handles the ENTER FOR DUPLICATE CLEAR TO BYPASS prompt.	SA
EnterAuthorizationErrorHelper	Handles state 6 in GSA. If the error handling mode is automatic, the error helper sends a hard coded 19 as the authorization code. If the error handling mode is default, the error helper waits for the operator to enter an authorization number.	GSA
Enter Error Helper	Assists with prompts that require an ENTER by the operator. In automatic mode, the simple case is just ENTER. In callback mode, a list of choices can be presented to select from.	ACE
ExpirationDateErrorHelper	Handles the ENTER EXPIRATION DATE prompt. For automatic or default error handling mode, the error helper extracts the expiration date from the TenderIdentifier. If none is provided, an exception is thrown.	ACE
FatalClearErrorHelper	Used for errors that are always non-recoverable. The error requires two clear keys to clear the error. In automatic error handling mode, the error helper sends the clear keys. In default error handling mode, the error handler waits for the operator to clear the error. In either mode, an exception is thrown after the error is cleared.	ACE, SA
FatalErrorHelper	Used for errors that are always non-recoverable. In automatic error handling mode, the error is cleared by the error helper. In default error handling mode, the error helper waits for the operator to clear the error. In either mode, an exception is thrown after the error is cleared.	ACE, GSA, SA
InformationalErrorHelper	Used for guidance messages that are informational in nature. In automatic error handling mode, the error helper clears the guidance. In default mode, the error helper waits for the operator to clear the guidance.	ACE, GSA, SA
InsertDocumentHelper	Used to handle the INSERT DOCUMENT FOR FRANKING prompt. In automatic and default error handling mode, the error helper waits for a document to be detected at the printer document station.	GSA
ItemPromptErrorHelper	Used during item entry sequences to handle prompts for additional data such as item price, item weight, and volume. In automatic error handling mode, the prompt is cleared that terminates the item entry. An exception is thrown to indicate to the client which argument must be added to the addItem method call. In default error handling mode, the error helper waits for the operator to satisfy the prompt, or cancel the item. If the operator cancels the item, an exception is thrown.	ACE, SA

Table 10. Error helper classes (continued)

Error helper key	Error helper behavior	Implementation classes available for applications
OriginalSalespersonRequired ErrorHelper	Used to handle state 20 ORIGINAL SALESPERSON REQUIRED. In both automatic and default error handling modes, the error helper attempts to send the original salesperson ID key sequence.	GSA
OverrideErrorHelper	Used when the application is giving guidance that a manager or operator override is required to complete the operation. In automatic error handling mode, the error helper checks to see if the Automation Provider is configured to perform the override automatically. If configured for automatic override, the error helper clears the guidance and performs the override. In default error handling mode, the error helper waits for the operator to clear the guidance and perform the override, or cancel the operation. If the operator cancels the operation, an exception is thrown.	ACE, GSA, SA
Pinpad Error Helper	Handles pin pad errors. In automatic mode, there is a check made to see if the configuration allows for electronic signature to be bypassed, if so it will bypass it. In callback mode the prompt will be sent to the application to be handled.	ACE
PriceRequiredErrorHelper	Used to handle the A109 PRICE IS REQUIRED prompt. In automatic or default error handling modes, the error helper sends the price sequence if a price was included in the ItemIdentifier. If no price was included in the Identifier and the error helper is in automatic mode, an exception is thrown. If no price was included in the Identifier and the error helper is in default mode, the error helper waits for the operator to enter a price.	GSA
PrintBypassErrorHelper	Used to handle the ENTER TO PRINT OR CLEAR TO BYPASS prompt. If the error handling mode is automatic, the error helper waits for the document to be inserted, then sends the ENTER key to print the document. If the error handling mode is default, the error helper waits for the operator to press ENTER or CLEAR.	GSA
PrinterErrorHelper	Used to handle printer errors. The error helper enters a loop, waiting for the printer problem to be corrected.	ACE, GSA, SA
PrinterPaperOutErrorHelper	Used to handle *out of paper* printer errors. The error helper waits for the printer error to be resolved by the operator.	GSA, SA
QuantityErrorHelper	Used to handle state 49 where a quantity is required for an item. If the error handling mode is automatic or default, the error helper will use the quantity provided in the ItemIdentifier. If no quantity was provided in the ItemIdentifier, an exception is thrown.	GSA
QuantityOrPriceErrorHelper	Used to handle the a prompt for quantity or price. If the error handling mode is set to automatic the error helper sends the price or quantity sequence, if the price or quantity was included in the ItemIdentifier. If not, an exception is thrown. If the error handler mode is default, the error helper waits for the operator to satisfy the prompt for price or quantity.	GSA
RainCheckErrorHelper	This error helper handles rain check specific errors. When handling the 'ENTER = FINAL PRICE CLEAR = PRICE PER LB' prompt, the error helper checks to see if the client input a weight (keyed or scale). If it did, the error helper sends the 'clear' key and then inputs the weight. Otherwise, it sends the enter key. In callback mode, the prompt will be sent to the client application to be handled.	ACE
RemoveDocumentHelper	Used to handle the REMOVE DOCUMENT prompt. In both automatic and default error handling modes, the error helper waits for the document to be removed from the document insert station of the printer.	GSA
RemoveReceiptErrorHelper	Used to handle the REMOVE RECEIPT FOR CUSTOMER prompt. • In automatic error handling mode, the error helper clears the error. • In default error handling mode, the error helper waits for the operator to clear the error.	SA

Table 10. Error helper classes (continued)

Error helper key	Error helper behavior	Implementation classes available for applications
SAErrorHelperSignature	Used to handle the VERIFY SIGNATURE prompt. If the error handling mode is automatic, the error helper sends the key sequence to verify the signature. If the error handling mode is default, the error helper waits for the operator to enter the key sequence to verify the signature.	SA
TerminalDisabledErrorHelper	Used to handle the TERMINAL DISABLED message. In automatic error handling mode, the error helper clears the error, then throws an exception. In default error handling mode, the error helper waits for the operator to clear the error, then an exception is thrown.	GSA
UserData Error Helper	Handles requests from ACE for extra user specified data. These prompts are generated when the user has enabled user data in ACE personalization. In automatic mode, if additional data was sent then it will be used. Otherwise the error is fatal.	ACE
VoucherNumberErrorHelper	Used to handle the ENTER VOUCHER NUMBER prompt. For automatic or default error handling mode, the error helper extracts the voucher number from the TenderIdentifier. If none is provided, an exception is thrown.	ACE
ZipCode Error Helper	Handles requests for a zip code. The prompt is just cleared in automatic mode which may lead to the failure of the request that generated the prompt. In callback mode, a request is sent to get a zip code.	ACE

Data Provider

The purpose of the Data Provider is to publish events that represent changes in the POS application, and to maintain current property values that represent the current state of the POS application. Clients can register to listen for Data Provider events such as TransactionStatusEvents, ItemSalesEvents, TransactionTotalsEvents, OperatorEvents, and so on. Clients can also query or set individual property values. When a Data Provider property value is changed, an (AEFPropertyChangeEvent) is sent to any registered listeners.

Properties

Data Provider properties are key/value pairs. Each property has a category (type string), a property name (type string), and a value (usually type string).

The category can be any string value, but there is a set of predefined categories listed in Table 11.

Table 11. Data Provider Property categories

Category	Category Name	Category Name Constant	Contains Properties
ConnectionChangeProperties	CONNECTION-CHANGE$_1$	ConnectionChange-Properties.CATEGORY$_1$	Individual properties from the last ConnectionChangeEvent.
CouponProperties	COUPON	CouponProperties.CATEGORY	Individual properties from the last CouponEvent.
CustomerProperties	CUSTOMER	CustomerProperties.CATEGORY	Individual properties from the last CustomerEvent.
DiscountProperties	DISCOUNT	DiscountProperties.CATEGORY	Individual properties from the last ItemSalesEvent.
FuelOptionsProperties			Properties related to fuel options personalization of the POS application.
ItemSalesProperties	itemSales	ItemSalesProperties.CATEGORY	Individual properties from the last ItemSalesEvent.
LineItemProperties			Individual properties from the last line item event (of any type).

Table 11. Data Provider Property categories (continued)

Category	Category Name	Category Name Constant	Contains Properties
OperatorAuthorizationProperties	opAuth	OperatorAuthorizationProperties. CATEGORY	Each property in this category indicates a POS function. The property value is true or false to indicate whether the operator is authorized to execute the function.
OptionsProperties	OPTIONS	OptionsProperties.CATEGORY	Properties related to POS application options settings.
PipeDataProperties	DATA_FOR_PIPE	PipeDataProperties.CATEGORY	Individual properties related to data that is written on a pipe to send to the POS application.
PointsProperties	POINTS	PointsProperties.CATEGORY	Individual properties from the last PointsEvent.
POSDataProperties			
POSDeviceProperties	POS_DEVICE	POSDeviceProperties.CATEGORY	Properties related to POS I/O devices, such as the contents of the line displays, last labels scanned, cash drawer state, current application state and substate, and so on.
PreAuthProperties	Generic	PreAuthProperties.CATEGORY	Individual properties from the last PreAuthResponseEvent
StoreOptionsProperties			Properties related to store options personalization of the POS application.
StoreProperties	STORE	StoreProperties.CATEGORY	Properties such as store name, division, address, phone numbers, and so on.
SuspendedFuelKioskTransaction SummaryProperties			Individual properties from the last SuspendedFuelKiosk TransactionSummaryEvent.
TenderProperties	TENDER	TenderProperties.CATEGORY	Properties related to the last tender taken such as type, variety, description, amount, and so on.
TerminalOptionsProperties			Properties related to terminal options personalization of the POS application.
TransactionStatusProperties	transactionStatus	TransactionStatusProperties. CATEGORY	Properties such as transaction id, date, time, category, voided indicator, discounts allowed indicator, void allowed indicator.
TransactionTotalsProperties	transactionTotals	TransactionTotalsProperties.CATEGORY	Properties containing the transaction totals such as transaction amount, balance due, change due, tax amount, subtotal, and so on.
UserTLogEntryProperties	userTLogEntry	UserTLogEntryProperties.CATEGORY	Individual properties from the last UserTLogEntryEvent.
WorkstationStatusProperties	WORKSTATION _STATUS	WorkstationStatusProperties. CATEGORY	Properties such as terminal number, transaction number, transaction in progress indicators, training mode indicator, offline indicator, and so on.
TransactionReplayComplete EventProperties	transactionReplay Complete	TransactionReplayComplete Event.CATEGORY	Properties that indicate the date, time, and transaction ID of the last transaction that was recovered, retrieved, or transferred.
[1]The hyphen is **not** a part of the category name or category name constant.			

Figure 16 on page 42 illustrates the inheritance hierarchy of the Data Provider property interfaces.

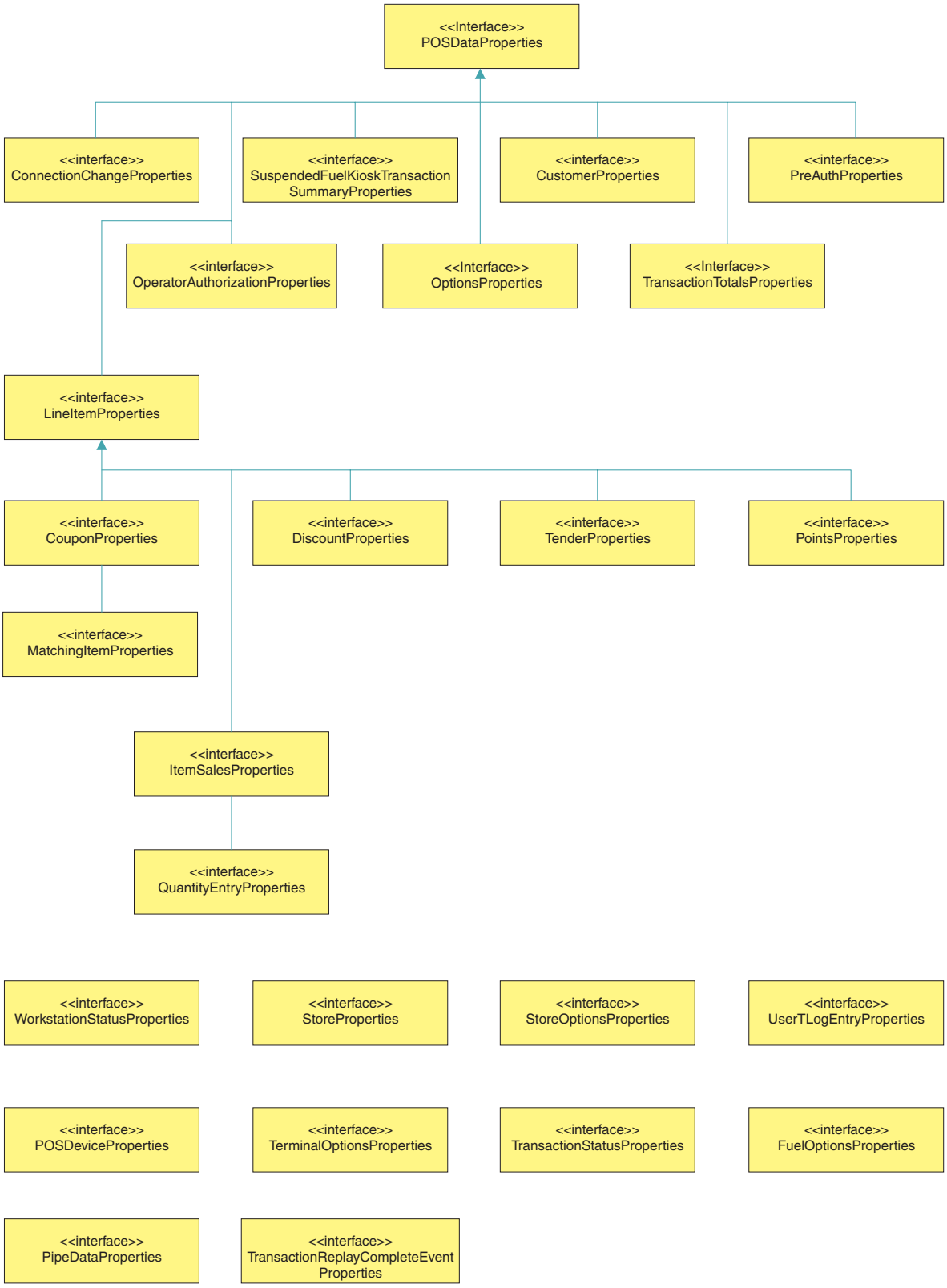

Figure 16. Data Provider property hierarchy

Event and listener types

The Data Provider supports a number of event and listener types. Figure 17 on page 44 illustrates the event type hierarchy.

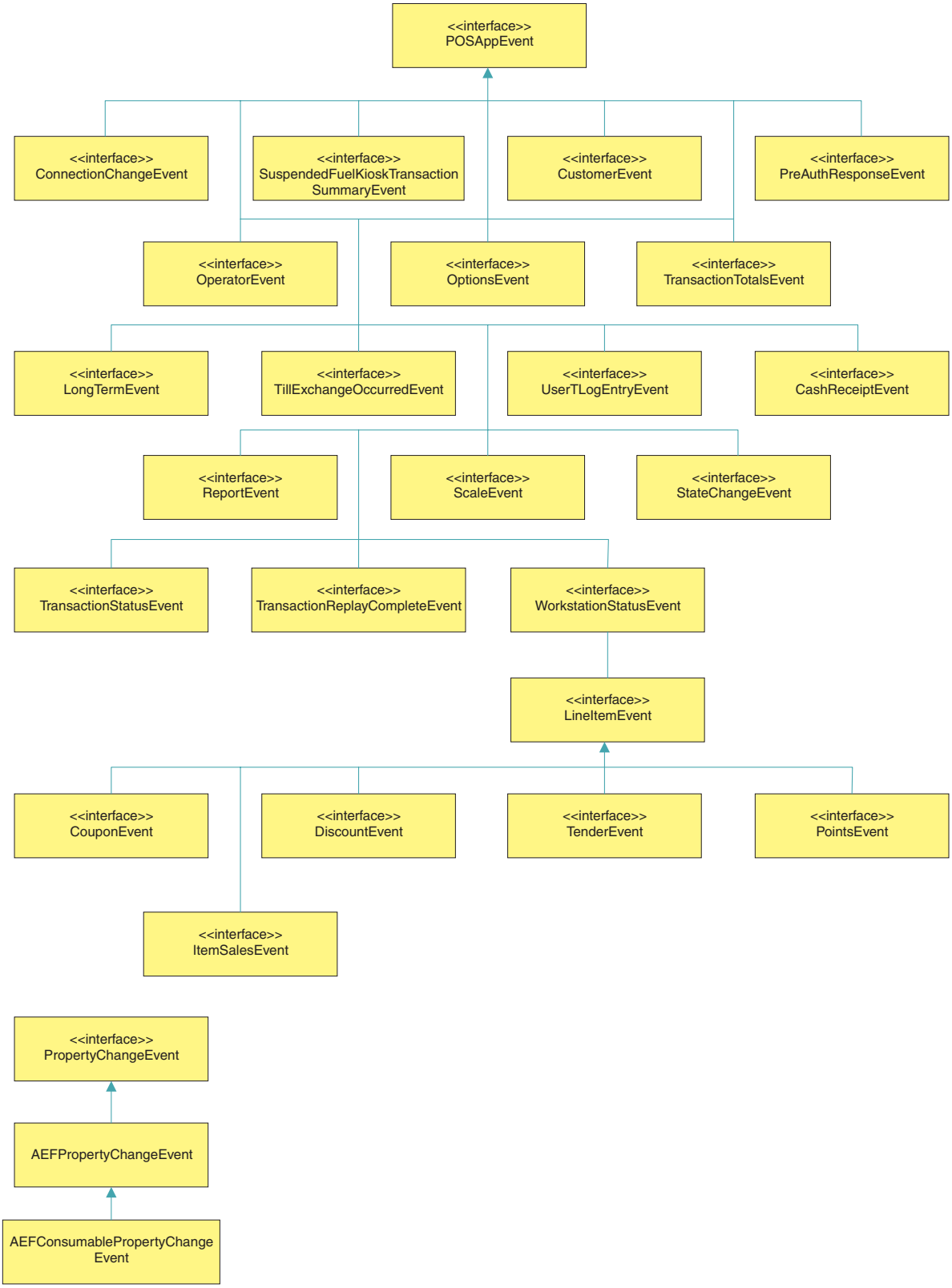

Figure 17. POSDataProvider event hierarchy

Each of the event types shown in Figure 17 is paired with a corresponding listener interface. The event and listener types are discussed below.

AEFPropertyChangeEvent

The AEFPropertyChangeEvent is triggered when an individual Data Provider property changes value. The event includes the property category, property name, old value, and new value. See Appendix E, "Data Provider properties," on page 207 for the properties supported by the Data Provider.

AEFPropertyChangeListener

To receive AEFPropertyChangeEvents, a client must implement the AEFPropertyChangeListener interface, and register with the POSDataProvider.addAEFPropertyChangeListener method.

AEFDateTimeListener

The AEFDateTimeListener provides an interface for receiving updates of the date and time. There are properties related to the time updates in config.properties. The default values are listed below:

```
# Specifies how often the AEFUtilties class will notify its listeners of time updates.
# These time updates are used to help remote clients maintain the same time as the
# controller they are attached to. Time is in milliseconds.
general.time.update.interval=900000

# Specifies how often the AEFUtilties class will check to see if the time has changed
# in a fashion that requires all listeners to update their time. This value needs to
# be between 0 and 50 seconds. Time is specified in milliseconds.
general.time.check.interval=10000

# Specified the number of clients to be notified of the current time per second when
# a time update is sent out.
general.time.update.throttle.limit=5
```

Figure 18. AEFDateTimeListener default values

POSAppEvent

The POSAppEvent is the base class for the Data Provider event interfaces. The event contains information such as the terminal number, event type, and property change category.

GenericEventListener

GenericEventListener is a general interface for listeners of POSAppEvents generated by the POSDataProvider API. It is used primarily to support extensions to the event listener API where a specific listener interface is not needed.

The following is a sample of code using the GenericEventListener:

```
SampleClass()
{
  try
  {
      genericProxy = new GenericListenerProxy(data, this);
  }
  catch (RemoteException re)
  {
     System.err.println("Remote exception adding listener - Exception=" + re);
  }
  catch (AEFException ae)
  {
     System.err.println("AEF exception adding listener - Exception=" + ae);
  }
}

public void eventOccurred(POSAppEvent evt) throws RemoteException
{
  if(evt instanceof TenderEvent||
     evt instanceof ItemSalesEvent||
     evt instanceof CouponEvent)
  {
     System.err.println("These events affect the sales transaction total");
  }
}
```

Figure 19. GenericEventListener sample code

POSAppEventListener

The POSAppEventListener interface is a base type for the other listeners in this section. Table 12 lists the event and corresponding listener pairs.

Table 12. POSAppEventListener events and corresponding listener pairs

Event name	Event trigger	Listener to implement	Method to register for event on POSDataProvider
AEFDateTime	The AEFSession detects an abnormal time change in the system.	AEFDateTimeListener	addAEFDateTimeListener
CashReceiptEvent	POS application sends print lines to the cash receipt printer	CashReceiptListener	addCashReceiptListener
CouponEvent	A store or manufacturer coupon is added to or voided from a transaction.	CouponListener	addCouponListener
CustomerEvent	A customer loyalty number is added to a transaction.	CustomerListener	addCustomerListener
DiscountEvent	A loyalty customer (with a discount code in their record) has been added to a transaction, and subsequently a transaction total has been requested.	DiscountListener	addDiscountListener
ItemSalesEvent	An item is added to or voided from a transaction.	ItemSalesListener	addItemSalesListener
OperatorEvent	An operator has signed on to the terminal.	OperatorListener	addOperatorListener
OptionsEvent	POS application loads a set of personalization options	OptionsListener	addOptionsListener

Table 12. POSAppEventListener events and corresponding listener pairs (continued)

Event name	Event trigger	Listener to implement	Method to register for event on POSDataProvider
PointsEvent	Customer loyalty points were awarded, redeemed, or voided in a transaction.	PointsListener	addPointsListener
ReportEvent	Report has been generated from the register, such as Open Transaction Report in ACE.	ReportListener	addReportListener
ScaleEvent	The scale connected to the register has changed (item placed on, or removed from scale).	ScaleListener	addScaleListener
StateChangeEvent	POSApplication transitions between states. For 4690 POS applications, the state is a number corresponding to the application's state table as processed by the I/O processor.	StateChangeListener	addStateChangeListener
TenderEvent	A tender has been added to or voided from a transaction.	TenderListener	addTenderListener
TransactionStatusEvent	A specific change in the state of the transaction has occurred such as being started, voided, suspended, or ended.	TransactionStatusListener	addTransactionStatusListener
TransactionTotalsEvent	Running totals within a transaction have been updated.	TransactionTotalsListener	addTransactionTotalsListener
WorkstationStatusEvent	Indicates a specific change in a POS register.	WorkstationStatusListener	addWorkstationStatusListener
TransactionReplayCompleteEvent	Indicates that a transaction recovery, retrieve, or transfer has completed.	TransactionReplayComplete EventListener	addTransactionReplayComplete EventListener

Extending POSAppEvents

This section provides a general overview of POSAppEvent objects and provides an introduction to their usage. It also provides information needed to utilize the extension mechanism provided by POSAppEventElement that is the parent class of POSAppEvent. POSAppEventElement provides a generalized event element container for additional event attributes and sub-elements. This mechanism can be used to add user data into the XML messages sent from the POS application, and have them automatically included in the relevant POSAppEvent.

POSAppEvent objects are created by parsing the XML messages provided from the base POS application. For example, a CustomerEvent object is the result of parsing XML defined by the <*customer*> XML message. See "Customer event" on page 249. The type of event object to be created is defined by the following line in the classes.properties file:

```
customer=com.ibm.retail.AEF.event.CustomerEventImpl
```

By default, the XML element tag is used as a key to lookup the event class in classes.properties. If no entry is found, then the POSAppEventImpl class is used.

The CustomerEventImpl class is a subclass of POSAppEventImpl and implements an API unique for customer data (for example, customer.getID returns the customer loyalty number). Although the

CustomerEvent interface provides a set of methods that cover the basics of customer data, there may be cases where additional data is required. In this case the more general interface provided by POSAppEventElement can be used.

For example, suppose a new attribute foo is needed in the customer event. The POS application can modify the XML sent to include this attribute in the customer XML:

```
<customer id="90071234566" foo="820515" />
```

See "Sending additional data to an event" on page 114 for information about how to modify the XML messages sent to the AEF by the POS application.

Although no method for getFoo() is provided in the CustomerEvent interface, the value of the attribute, foo, is available by using the getProperty method of the POSAppEventElement interface inherited by CustomerEvent. An example of this method is shown as follows:

```
public void customerCardEntered(CustomerEvent evt) throws RemoteException
   {
      String foo = (String) (evt.getProperty("foo"));
      if (foo != null)
      {
         System.err.println("The value of foo is " + foo);
      }
      else
      {
         System.err.println("No foo in this event!");
      }
   }
```

If foo is a complex element rather than a simple attribute, a <foo> element may be included in the XML. In this case, the AEF attempts to create an object of the appropriate type by referencing the userclasses.properties file for the correct classname. The class must be a subclass of the POSAppEventElementImpl class. If no entry is found, then an object of type POSAppEventElement is created by default. For example, if the XML sent from the POS application is:

```
<customer id="90071234566">
  <foo name="myfoo" attribute1="some more data" isFoo="true"  />
</customer>
```

The following code provides access to the attributes of the <foo> element:

```
public void customerCardEntered(CustomerEvent evt) throws RemoteException
  {
     Object element = (POSAppEventElement) (evt.getProperty("foo"));
     if (element != null)
     {
        if (element instanceof Collection)
        {
           System.err.println("We have many foos");
        }
     }
     else
     {
        POSAppEventElement foo = (POSAppEventElement) element;
        System.err.println("The value of foo attribute1 is " +
                           foo.getProperty("attribute1"));
     }
  }
```

Listener considerations

The POSDataProvider maintains a queue of events that must be dispatched to listeners. The listener event notification mechanism is synchronous. This is of special concern for clients of virtual sessions. If a client

delays or blocks the POSDataProvider dispatch thread while processing an event, then event dispatching for all virtual sessions is delayed or blocked until the client returns from handling its event.

When handling a data provider event, the client listener should process the event as quickly as possible. If the processing is not trivial, the client should handle the event on another thread instead of the data provider dispatch thread.

To make efficient use of the data provider dispatch thread, the AEF includes proxy listener classes that may be used by the clients. The proxy performs event queuing and thread context switching so that the events are automatically handled on a different thread than the data provider event dispatch thread.

Clients should use the proxy listeners when possible, especially when the client is remote.

For each of the data provider event listener classes, there is a corresponding proxy class as listed in Table 13.

Table 13. Data provider event listener proxies

Regular Listener Class	**Proxy Listener Class**
AEFDateTimeListener	(DateTimeListenerProxy)
CashReceiptListener	(CashReceiptListenerProxy)
ConnectionChangeListener	(ConnectionChangeListenerProxy)
CouponListener	(CouponListenerProxy)
CustomerListener	(CustomerListenerProxy)
DiscountListener	(DiscountListenerProxy)
POSAppEventListener	(GenericListenerProxy)
ItemSalesListener	(ItemSalesListenerProxy)
OperatorListener	(OperatorListenerProxy)
OptionsListener	(OptionsListenerProxy)
PointsListener	(PointsListenerProxy)
ReportListener	(ReportListenerProxy)
ScaleListener	(ScaleListenerProxy)
SessionOwnershipListener	(SessionOwnershipListenerProxy)
SessionStatusListener	(SessionStatusListenerProxy)
StateChangeListener	(StateChangeListenerProxy)
TenderListener	(TenderListenerProxy)
TillExchangeOccurredListener	(TillExchangeOccurredListenerProxy)
TransactionStatusListener	(TransactionStatusListenerProxy)
TransactionTotalsListener	(TransactionTotalsListenerProxy)
UserTLogEntryListener	(UserTLogEntryListenerProxy)
WorkstationStatusListener	(WorkstationStatusListenerProxy)
TransactionReplayCompleteEventListener	(TransactionReplayCompleteEventListenerProxy)

See “Subscribing to a Data Provider event” on page 112 for an example of subscribing to a Data Provider event.

Service Provider

The service provider API defines a standard set of messages that form a communication language between a point of sale application and any services that want to provide additional function to it. The service provider API also provides a Java interface that defines the methods a service developer must use to integrate with the point of sale application through Store Integrator.

Store Integrator transports messages between a service and the point of sale application via the SILocalServiceBus. The SILocalServiceBus is a Java based messaging bus that transports XML messages using direct Java invocations.

The following diagram describes the flow of events between the POS application and a service via the SILocalServiceBus. Event processing in the AEF occurs on an event queue thread that is separated from the main input thread.

Legend:

ASC = ApplicationServiceConnector

ADC = ApplicationDataConnector

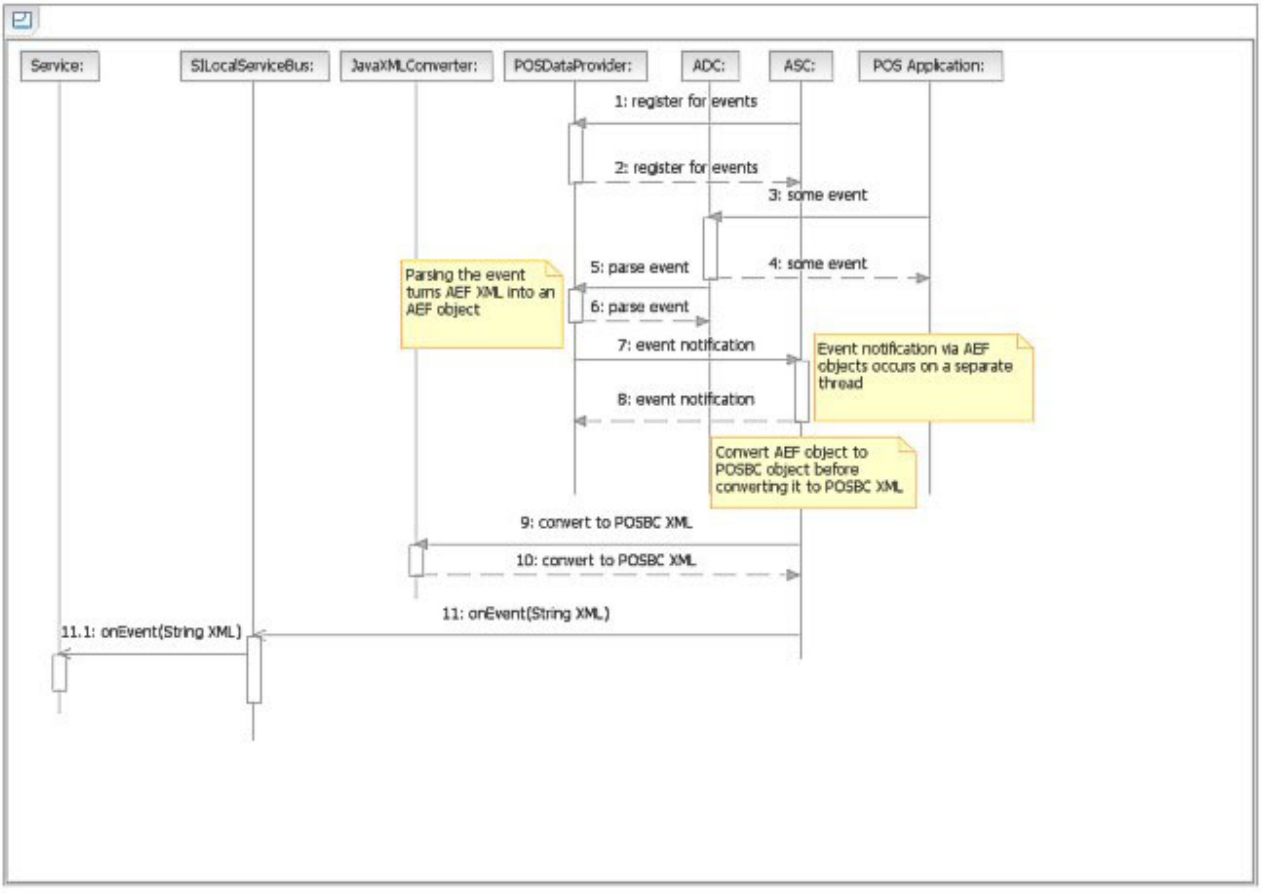

Figure 20. Flow of events

The following diagram shows the flow of requests and responses between a POS application and a service.

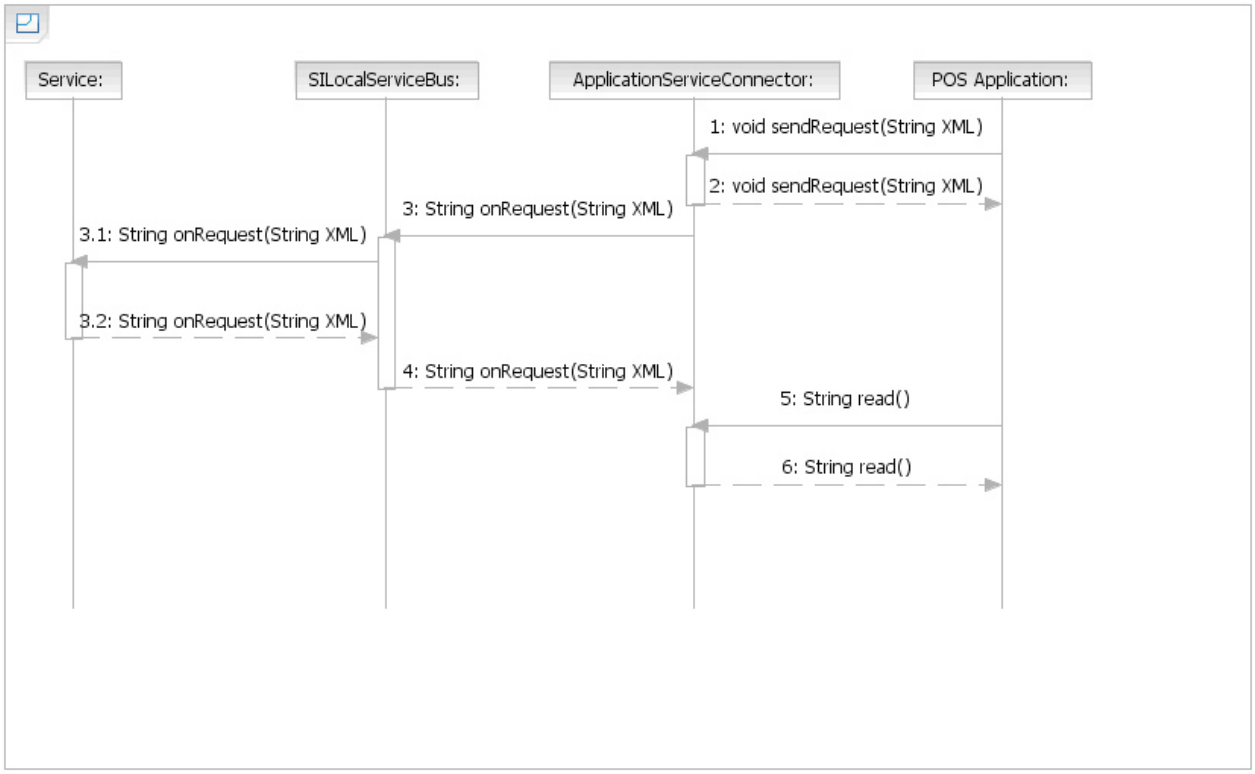

Figure 21. Flow of requests

POSProvider/POSServer API

The POSProvider provides a simple client interface for sending requests to a point-of-sale (POS) system. It includes a basic set of operations to perform the tasks required to process a sales order through the point-of-sale system. It enables the client to perform the following functions:

- Check the status of the point-of-sale system
- Request unit price information
- Request extended prices and transaction totals for an order
- Pay for the order
- Confirm responsibility for reversing a charged payment

The POSProvider uses *request* and *result* objects to contain the client input and output needed to process a request. The request objects are extensible container objects that are passed to the point-of-sale system. They are intended to provide the point-of-sale system the input needed to complete a specific POS operation. For example, a request to obtain unit prices utilizes a PriceRequest object as input to the method call to get the base item prices.

The result objects are similar to the request objects and provide a flexible means of returning information from the point-of-sale to the client application, for example, pricing results are returned in a PriceResult object.

Each request and result can have additional parameters added to it as an extension mechanism. A method is supplied to allow any name/value pair to be added to a request. This allows for additional data to be supplied by the client if necessary (for example to respond to new prompts) without changing the POSProvider interface.

Object creation for the POSProvider is provided through a factory interface. The POSProviderFactory provides the interface to instantiate instances of the POSProvider and the objects required to create and process a request through the POSProvider interface.

Payment integrity is introduced using a simple protocol. See "Payment integrity" on page 57 to determine whether the client or the server is responsible for *reversing* a charge that has been made to a customer in the event of a communication or hardware failure after payment has been authorized but before the client has been informed.

Figure 22 on page 53 and Figure 23 on page 54 illustrate retrieving the POSProvider from the POSProviderFactory and using it to process a sales transaction. This function is one of several provided by the POSProvider.

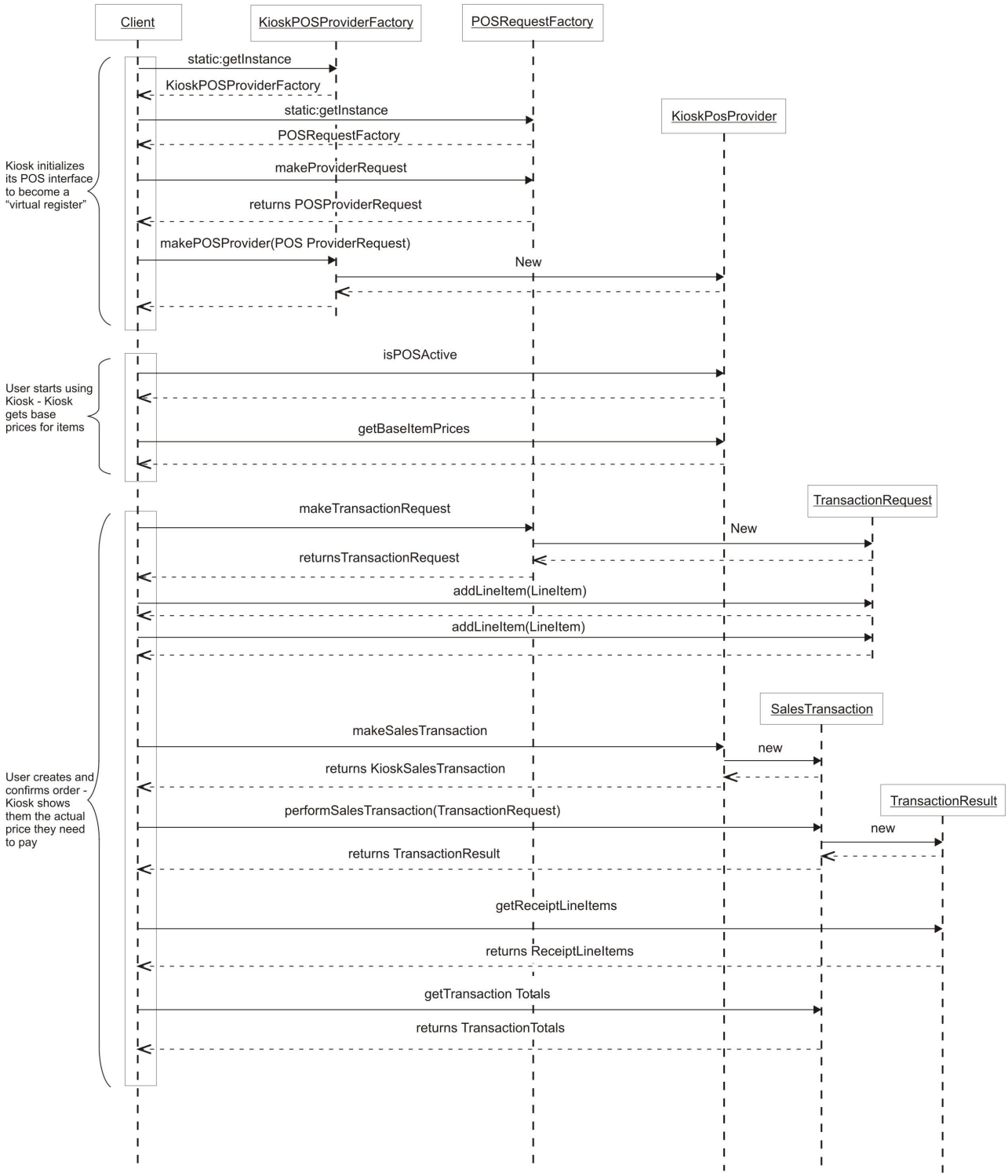

Figure 22. Retrieving the POSProvider from the POSProviderFactory

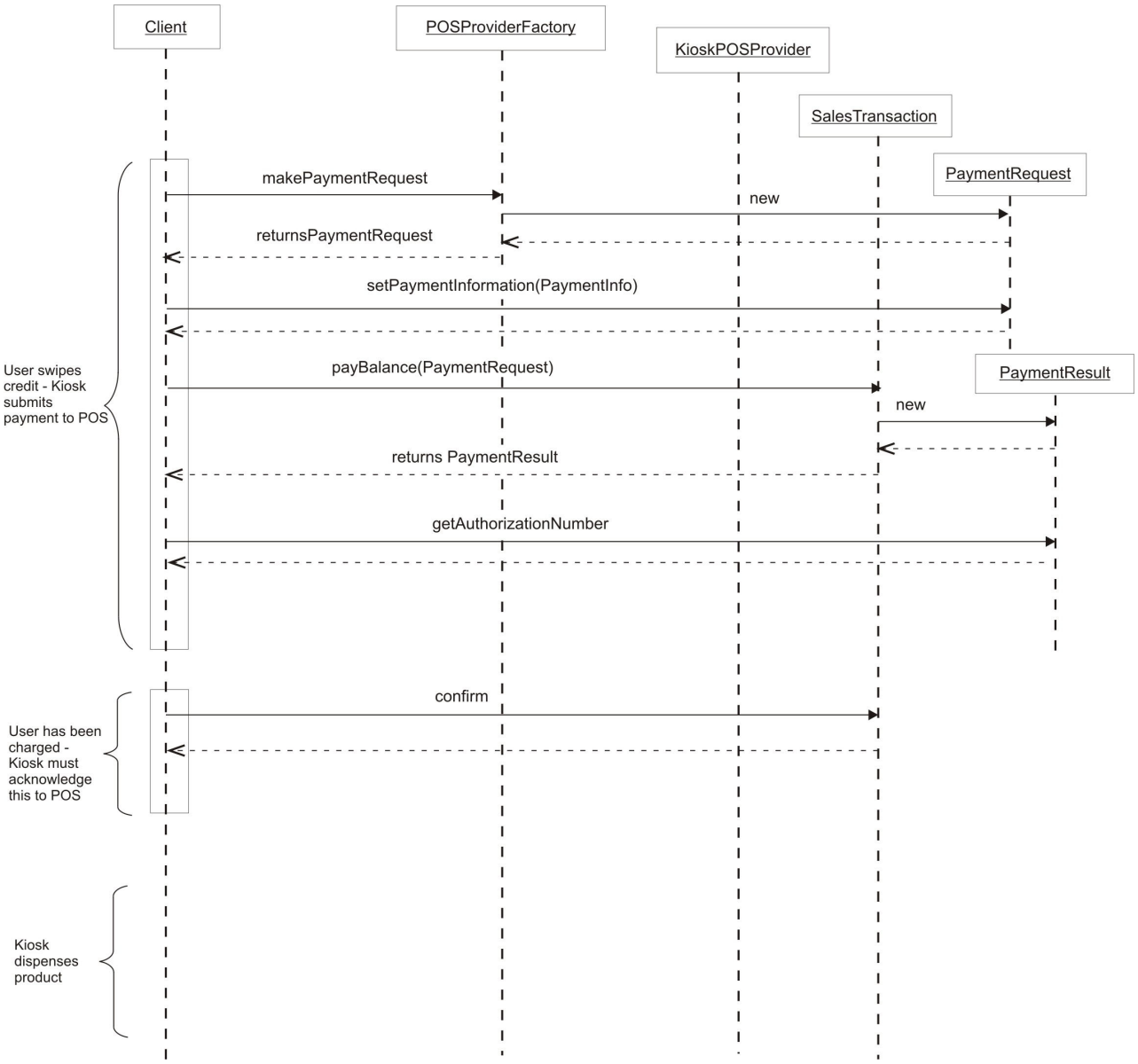

Figure 23. Using the POSProvider to process a sales transaction

POSProvider

Provides the common super-interface to define specific modes of accessing the point-of-sale system.

POSProvider instances allow client access to the point-of-sale system through high-level functions that accept POSRequest objects as input and return POSResult objects as output.

POSProvider defines the service methods to determine the status of the point-of-sale system and record exceptions in the POS exception log.

An instance of POSProvider is obtained through an implementation of the POSProviderFactory interface. A ProviderRequest object defines the parameters used by the factory to instantiate a POSProvider and connect to the point-of-sale system. These parameters include the terminal number or a range of terminal numbers used by the client to access the point-of-sale system.

After an instance of POSProvider is obtained, it may be used to submit requests to the point-of-sale system. After the client has completed its processing, it should call the release method before shutting down. The release method releases the point-of-sale resources and makes them available for subsequent access through the POSProviderFactory.

POSProviderFactory

Provides a factory API to obtain instances of POSProvider objects. An implementation of the POSProviderFactory utilizes the factory pattern to create and return instances of provider objects. It is responsible for creating the POSProvider objects that provide access to the services available on the point-of-sale system.

A POSProvider object is usually obtained by sending a ProviderRequest to the factory through the make POSProvider method. More advanced users that want to use the POSServer directly can call the get POSServerFactory and the getBackupPOSServerFactory methods to obtain the factories for the POSServers running on those controllers.

KioskPOSServer

Provides the common interface to define specific modes of accessing the point-of-sale system for Kiosk implementations of the POSServer.

KioskPOSServer defines the service methods to determine the status of the point-of-sale system, do simple pricing queries, and record exceptions in the POS exception log.

An instance of KioskPOSServer is obtained through an implementation of the KioskPOSServerFactory interface. A ProviderRequest object defines the parameters used by the factory to instantiate a KioskPOSServer and connect to the point-of-sale system. These parameters include the terminal number or a range of terminal numbers used by the client to access the point-of-sale system.

After an instance of KioskPOSServer is obtained, it may be used to submit requests to the point-of-sale system. An instance of KioskPOSServer may be used repeatedly by the POSProvider to submit these requests. After the POSProvider has completed its processing, it should call the release method before shutting down.

POSServerFactory

Provides an implementation of the factory pattern for POSServer objects. This implementation of the POSProviderFactory utilizes the factory pattern to create and return instances of POSServer objects. It is responsible for creating the POSServer objects that provide access to the services available on the point-of-sale system.

POSRequestFactory

Provides a factory API to obtain instances of objects needed to create requests for the POS. An implementation of the POSRequestFactory utilizes the factory pattern to create and return instances of request objects.

SalesTransaction

SalesTransaction provides an interface to perform point-of-sale operations with an active POS transaction.

POSInfo objects

The interfaces within the POSProvider/POSServer API provide a way to pass information to and from the POSServers. The POSInfo objects provide an extensible interface for exchanging information with the point-of-sale system using attributes. Extended attributes are contained as a collection of name-value pairs that can be easily stored and accessed.

PriceInfo objects

This interface contains price information about an item in the point-of-sale system. The PriceInfo objects contain all information relative to an item that was requested by the POSServer to the POS system. This interface is used for getting item descriptions and the regular price of an item and not used for the actual sale of an item.

PrintLine objects

This interface provides access to the line that was sent to the point-of-sale printer by the point-of-sale application. The line is available as filtered text along with format getter methods and also as raw text with the printer control characters embedded in the string. The filtered text string contains print line text including the spacing and blank lines, but printer attributes such as emphasized or double-high text are removed from the string and made available in getter methods. The raw text string contains an embedded representation of the printer control characters (escape sequences) that are used by the POS application to indicate text attributes such as bold text and font sizes.

ResultLineItem objects

This interface provides item identifier and quantity information for an item. This interface is used to send the information for an item that is being sold or returned. ResultLineItem objects also contain coupon information pertaining to the item being sold as well as the items identifier information.

CashReceiptImage objects

This interface provides an image of the point-of-sale register cash receipt. A cash receipt is produced by the point-of-sale register as the result of performing a transaction. The CashReceiptImage provides an object representation of the receipt.

POSRequest objects

POSRequest objects provide the information needed to process a client request for a point-of-sale operation. POSRequest objects are used for parameter passing with a POSProvider. The POSProvider uses the request object to determine the data needed to perform the point-of-sale operation provided by the provider API. Extensions to POSRequest may provide specific methods and instance data for the information required for a specific request. Additional data may be provided by the client through the more general setAttribute interface. The POSRequest interface shown in Figure 24 has several other interfaces that extend it to handle specific requests. These interfaces are:

- PaymentRequest
- PriceRequest
- ProviderRequest
- SalesRequest
- TransactionRequest

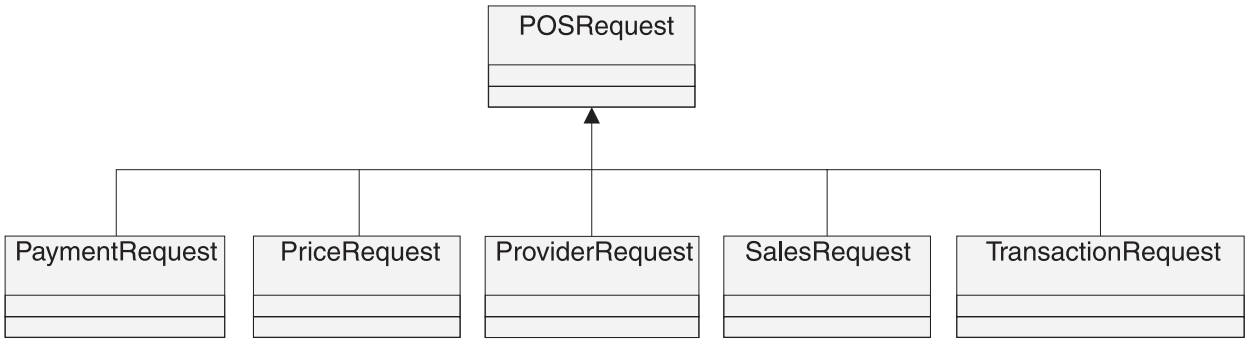

Figure 24. POSRequest objects flowchart

POSResult objects

POSResult objects contain information returned from the point-of-sale system. POSResult objects are used as return objects by POSProvider to contain the results of a point-of-sale operation. Extensions to POSResult may provide specific methods and instance data for the information required for a specific result. Additional data can be provided to the client through the more general setAttribute interface. The POSResult interface has several other interfaces shown in Figure 25 that extend its functionality to handle the following specific results:

- PaymentResult
- PriceResult
- SalesResult

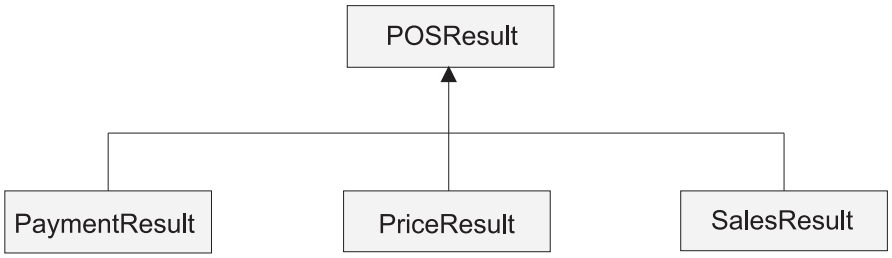

Figure 25. POSResult objects flowchart

Payment integrity

What happens if the client issues payment for an order and the customer's account has been successfully charged by the POS but then a failure occurs between the client and the controller? How does the client know whether or not to vend or deliver the goods to the customer? Whether or not the customer expects a refund? To help with these issues, the POSProvider uses a simple protocol to help determine whether it is the client or the responsibility of the POSServer to refund a customer. The protocol is as follows:

1. The client performs a sales order (performSales()), gets the total due in a result and pays for the balance using a credit card (payBalance()).
2. The POSServer pushes the tender through the POS and returns the authorization code and so on, for a successful authorization, back to the client. In the mean time it starts a confirm timer waiting for an acknowledgement by the client.
3. The client receives the payment result and immediately sends an acknowledgement: (confirm()).
4. When the POSServer receives the confirm(), it stops its confirm timer.
5. If for any reason the POSServer confirm timer expires, then the POSServer has not received payment confirmation, (because the client failed to authorize payment in time) and the POSServer starts a transaction refund procedure. Currently this procedure consists of writing a POS Exception log entry (see “POSServer Exception Log” on page 59) containing data, to give the retailer enough information to manually refund the transaction. A 4690 Application Event Log entry is also created to alert the retailer that a *transaction-to-be-refunded* entry has been made.

If the POSServer receives a client confirm and sends a response back, the POSServer is no longer responsible for refunding the transaction. The client must vend the goods or be responsible for refunding the customer. Conversely, if the client times out on a confirm response or gets a connection exception on the confirm, the POSServer should refund any charge made to the customer without the client’s intervention. What happens if the controller or POSServer crashes after a credit payment has been authorized but before a confirmation has been made? A checkpoint file (see “Checkpoint file” on page 59) is kept for each virtual terminal session that maintains key data about the transaction, payment authorization, and the state of the POSServer. That way, if the system crashes, when it starts up again, the POSServer startup routine can check any existing checkpoint files to determine if a POS Exception log entry needs to be made.

Client scenarios

A variety of payment/confirmation scenarios are shown below. Note that the italicized procedures such as *Client Initiated Refund* are shorthand references to procedures that are explained more fully in the subsequent Procedure Definition section. A further description of suggested client responses to exceptions is given in the Exceptions and Errors section.

1. **Payment Authorized**: Items were added to transaction, payment made and authorized, confirmation response received by client.
   - Customer has been charged by POS and the client has confirmed acknowledgement. The client is now responsible for dispensing goods to the customer. Now if there is any subsequent failure to dispense goods or the client determines that the customer requires reimbursement, it results in a *Client Initiated Refund*. (See "Procedure definitions.")
2. **Payment Exception**: Items were added to the transaction, a non-timeout and non-remote exception occurred while attempting a payment.
   - The customer has not been charged by the POS system, so the payment can be attempted again or the transaction can be cancelled.
3. **Confirm Exception**: Items were added to the transaction, a payment was made and authorized, but a non-timeout, non-remote exception occurred on the confirmation attempt. The customer has been charged by the POS system. The customer transaction will be rolled back by the POSServer (POSServerTransaction Refund Procedure). The client must assume that the customer was not charged.
4. **Payment Timeout/Connection Error**: Items were added to the transaction, and a timeout or remote exception occurred on the payment attempt.
   - The client does not know whether the customer was charged (because it is not known if the payment request or the response was lost) but the client can assume that the customer will be refunded because of either of the following conditions:
     a. Items were added to the transaction, and a timeout or remote exception occurred on the payment attempt.
     b. The POSServer or POS controller crashed, and upon restart the startup code checks for *Checkpoint files* (see "Checkpoint file" on page 59), which indicate that a payment was authorized, but a confirm was pending. If a checkpoint file is found, then the POSServer performs the *POSServer Transaction Refund* procedure.
5. **Confirm Timeout/Connection Error**: Items were added to the transaction, a payment was made and authorized, and then a timeout or remote exception occurs on the confirmation attempt.
   - The client knows that the customer was charged but does not know whether or not the confirm timer was stopped by the POSServer (because it is not known if the confirm request or the response was lost). However, the client should assume that the customer will be refunded. To ensure this happens, the POSProvider goes into a *POSProvider Transaction Refund* procedure. In the worst case, this might result in two Exception Log entries one from the POSProvider and one from the POSServer, but both are for the same transaction number.

Procedure definitions

This section describes procedures.

Client Initiated Refund

A customer gets a receipt containing dispensed line items that were not dispensed. The customer takes the receipt to the customer service desk for a refund. The receipt can contain the client transaction reference number plus the POS transaction number (from the TransactionResult) and should have special wording to indicate that the items could not be dispensed.

POSProvider Transaction Refund Procedure

The POSProvider attempts to log a POSException entry through the POSServer and loops until successful. The client is not able to execute another request until this is done.

POSServer Transaction Refund Procedure

The POSServer attempts to log a POSException entry on the controller. The client is not able to execute another request until this is done.

Checkpoint file

One Checkpoint file is kept for each virtual terminal session (such as $901.ckp for terminal 901). It is created at the start of a transaction and deleted at the end of a transaction. It is updated before a tender payment is performed and after the result is returned. The default location for the checkpoint files is M:\PosProv on the controller. The checkpoint file is an internal file used by the POSServer and should not be modified or deleted.

POSServer Exception Log

The POSServer Exception Log is implemented here as a file per exception (to facilitate deletion of files when processed). Because the files have long names for identification purposes, they are stored on the 4690's virtual drive, M:, in the M:\PosProv\Exceptions directory. It is used to keep track of critical POS exceptions (usually ones that affect the customer or the store financially). It will contain enough information about a transaction to enable the store manager to do one of the following:

- Use this information to verify that the transaction was tendered successfully using the point-of-sale's transaction log.
- Manually refund the amount to the customer.

An entry is written under the following conditions:

- The POSServer initialized after a crash, read the checkpoint file, and found that the state was *Pending Confirm*.
- The sales process was cancelled by the client instead of being confirmed after payment occurred.
- The POSProvider issued a request to the POSServer to log a POSException because it timed out on a confirm().

It is possible to have multiple entries for the same transaction that can be used as a fail-safe measure. The manager can check using the transaction numbers supplied in the entry.

The data shown in Table 14 is logged in a POS Exception Log Entry.

Table 14. POS Exception Log Entry data

Tag	Description
Log Date/Time	Date and time that this entry was logged.
ExceptionType	Reason for this Exception Log entry. Currently this can only be: Transaction Refund Needed.
Source	Definition of component that logged this error, such as: IBM.Si.PosProvider.Server or IBM.Si.PosProvider.Client.
Terminal #	Terminal number assigned to POSServer and AEF session.
Client TransactionID	The TransactionID assigned by the client when the POSServer transaction was created.
POS Transaction #	The transaction number assigned by the POS for this terminal transaction.
Transaction Date/Time	Date and Time that POS transaction was started (as returned by POS application). Should be the same as on the record posted to the TLOG.
State	The state of the transaction as kept by the POSServer.
EFT Authorization #	The Authorization number returned by the host.
EFT Seq #	The EPS Sequence number used by the EPS software for the message to the host (optional – it may not always be available).
Transaction Amount	The total transaction amount including tax.

Exceptions and errors

Table 15 describes an overview of the main operational exceptions and errors that might occur while using the POSProvider API. The comprehensive list and description of exceptions and errors is contained in the JavaDoc for the POSProvider API.

Table 15. Log levels with sample log entries

Exception	Extends ...	Reason
RemoteException	Exception	Connection error to the remote object (not a timeout).
POSException	Exception	General POSProvider or POSServer error.
TimeoutException	POSException	Network or application delay problem.
RequestException	POSException	Error with the data provided in the request.
ApplicationException	POSException	POS application exceptions like item not found, weight required, price required, limit checks. These are usually the results of an improperly configured POS system.
TenderException	ApplicationException	The POS could not authorize payment. The exception contains error information.
TransactionException	ApplicationException	An indication that an exception has occurred while trying to perform a sales order. This exception may contain multiple ItemExceptions.
ItemException	ApplicationException	An exception occurred trying to sell a single item.
TransactionInProgressException	IllegalStateException	Transaction is in progress. The request is in conflict with an ongoing transaction.
ProviderStateException	IllegalStateException	An indication that the POSProvider or POSServer was not ready to process a request.
TransactionStateException	IllegalStateException	An indication that the Transaction was not in a valid state to process a request. The request is not valid for the current state of the transaction. In general, this occurs, if a confirm() is issued for a transaction whose balance has not been satisfied.
RecoveryException	IllegalStateException	The POSProvider is not able to accept requests because the POSProvider is in error recovery state.

Client exception handling

Table 16 lists the exception types and the recommended response of the client.

Table 16. Exception types and response action categories

Exception	Response action (see Table 17 on page 61)
RemoteException	Health Check Loop
POSException	General POSProvider or POSServer error
TimeoutException	Health Check Loop
RequestException	Client Programming Error
ApplicationException	POS Configuration error
TenderException	EFT Operational Error
TransactionException	POS Configuration error
ItemException	POS Configuration error
TransactionInProgressException	Health Check Loop
ProviderStateException	Client Programming Error

Table 16. Exception types and response action categories (continued)

Exception	Response action (see Table 17)
TransactionStateException	Client Programming Error
RecoveryException	Client Programming Error

Table 17 lists the exception types and the recommended action for the client.

Table 17. Client action descriptions

Client action	Description
Health Check Loop	Retry isPosActive() periodically until true or limit is exceeded. If limit is exceeded, inform customer about loss of POS service, continue to retry isPOSActive(). A 2 minute limit is recommended. Cancel any current transaction and start again.
Client Programming Error	Log an error and if possible, correct and resubmit. Otherwise cancel transaction and try again.
POS Configuration Error	Log an error. Store personnel might need to take action for POS application exceptions such as; item not found, weight required, price required, or limit checks. These are usually the result of an improperly configured POS system. Note that the Transaction has been voided by the POSServer before returning an Application Exception.
EFT Operational Error	The POS could not authorize payment. The exception contains error information. The payBalance() can be resubmitted with a new card number if necessary. Otherwise cancel() the transaction and start again.

Configuration

Client configuration and POSProvider configuration are described in the following sections.

Client configuration

The following items need to be configured by the client and passed in when creating the POSProvider:

- Terminal number – This must be allocated in conjunction with the retailer.
- IP Address of controller – for RMI registry location. If null then POSProvider uses RMI broadcast.
- IP Port – If null then default is used.

Configuring the IP address and port values by the client is recommended; however, it is possible to set the primary and backup controller information in the POSProvider property configuration files. See "POSProvider configuration" for the property specifications.

POSProvider configuration

The following items are specified in a standard Java properties file (posprov.properties) that is included in the POSProvider jar file (posprov.jar). If no properties file is supplied then the default values are used.

posprov.pos.timeout = POS Timeout
Client timeout value used to prevent a POS hang. Should be long enough to accommodate for host payment delays. Typically this should be set to exceed the AEF timeout values. If this is exceeded it means the POS is hung. Default = 600 seconds (10 minutes).

posprov.confirm.timeout = Confirm timeout
The timeout value the POSServer uses to wait for a confirm request from the client. Default = 60 seconds (1 minute).

posprov.tender.timeout = Payment timeout
The timeout value used to wait for a credit authorization response. Default = 120 seconds (2 minutes)

posprov.failover.retry.timeout = Failover reconnect timeout
The timeout value used to test and see if a failover has occurred. This serves as a timeout for the reconnect process. Default = 10 seconds.

posprov.response.timeout = Failover timout
The wait time for a response from the server before attempting to fail over. Default = 30 seconds.

posprov.pri.ip.addr
This sets the default IP address for the controller that is used as the primary controller.

posprov.alt.ip.addr
This sets the default IP address for the controller that is used as the backup controller.

posprov.ip.port
This sets the default IP port for the controller that is used for the primary and backup controllers. Default = 12099.

These values can also be overridden by the client on the POSProvider request and passed to the POSProvider on creation.

The properties file can be overridden by placing a copy of an alternative posprov.properties file in the classpath specified for the POSProvider or the POSServer ahead of the posprov.jar in the classpath.

POS Business Component

The POS Business Component (POSBC) interface extends the capabilities of AEF by providing a standard, XML-based messaging interface and a Java-based equivalent. The POSBC environment is a distributed environment consisting of the system unit hosting the POSBC (such as a pump controller in a fuel solution), and a 4690 controller running the Client Session Server (CSS) and hosting one or more AEF virtual sessions (terminals).

The POSBC XML/Java messaging interface provides the following functions:

Table 18. Functions of POSBC XML/Java messaging

Function	Description
Callback	A mechanism for allowing the client to directly respond to requests from the Point of Sale Application.
CancelAction	Cancels the current action in progress.
ContinueAction	Causes the current action to move on to the next logical processing state. The POSBC now sends options for the store during initialization.
Initialize	Initializes a POSBC instance. The POSBC now sends options for the store during initialization.
Terminate	Ends the instance of the POSBC. You can end the POSBC instance no matter its state.
ResetTerminal	Resets the POSBC.
SignOn	Signs on to a POSBC instance.
SignOff	Signs off of a POSBC instance.
InitializePrinter	Reconfigures the printer if it is not already initialized.
QueryStatus	Obtains the status of the POSBC instance.
AddItem	Adds an item to the current sales transaction.
VoidLastItem	Voids the last item that was added to the current sales transaction.
AddFuelSale	Adds a fuel item and any associated coupon items to a sales transaction.
AddCustomer	Adds a customer loyalty ID to the current sales transaction.

Table 18. Functions of POSBC XML/Java messaging (continued)

Function	Description
AddCustomerBirthdate	Adds a customer's birth date to the current sales transaction.
AddCoupon	Adds a coupon to the current sales transaction.
PriceVerify	Performs an item price verification.
GetTotals	Requests a final transaction total of the current sales transaction.
GetFoodStampTotal	Requests a food stamp total of the current sales transaction.
AddTender	Requests that a tender be added to the current transaction.
PreAuth	Requests that a pre-authorized tender approval be created, reversed or voided.
PreAuthCompletion	Requests that a previously authorized request be added reversed or voided.
Suspend	Requests that the current transaction be suspended.
VoidTransaction	Requests that the current transaction be voided.
ReportStatusEvents	Sends a series of POSBCStatusEvents describing the current state of the printer.
AddReceiptLines	Adds lines to the virtual receipt.
AddRegESignature	Adds the RegElines with merged signature to the virtual receipt.
RemoveReceiptLines	Removes lines from the virtual receipt.
RemoveReceipts	Removes whole virtual receipts.
Print	Requests that print data be sent to the printer.
ArchiveCurrentReceipts	Saves all of the current receipts.
PrintCurrentReceipts	Prints all of the current receipts.
ReplayCurrentReceipts	Requests a collection of the receipt lines from one of the current specified receipts.
PrintArchivedReceipts	Prints the specified archived receipts.
PurgeArchivedReceipts	Removes the archived receipts.
ReprintReceipts	Reprints receipts.
GenerateGiftReceipt	Generates gift receipts at the terminal application.
PrintGiftReceipt	Prints gift receipts.
LogException	Logs a user exception within POSBC and within the POS application, if available.
LogUserTLogData	Logs a user transaction log string to the POS application transaction log.
Resume	A request to resume/retrieve a suspend transaction.

JIFServer

The JIFServer is the underlying message transport implementation. The server listens on a configurable ServerSocket for all incoming and outgoing XML Business Component (BC) messages. The JIFServer manages all the BC connections as well as all their messages. Each connection receives a thread to listen for messages, and a new thread is spawned for each message received. The use of separate threads allows for parallel processing of BC connections, and concurrent messages/events from a single connection. The server maintains a mapping of connections to BCs so that it can direct outgoing messages to the correct connection.

BC messages are namespace-aware XML messages with additional headers built using the soeps message protocol.

The following is an example of a soeps header:

```
soeps~Session-Id=301|Message-Type=REQ|~
```

The JIFServer does not inspect the message content, but simply uses the soeps header to determine what to do with the message. The required soeps headers fields are:

Table 19. Required soeps header fields

Header field	Values
Session-Id	The identifier of the business component that is the message recipient. **Note:** Each BC instance needs a distinct session id so messages are routed properly.
Message-Type	EVENT – an asynchronous event REQ – a synchronous request message RESP – a response to a synchronous request

Message Processing

The JIFServer contains a BusinessComponentListener for processing BC messages. When the configured listener is instantiated, it is passed a reference to a JifMsgHandler interface that it will use to send outgoing messages back through the JIFServer.

In addition to passing incoming requests to the listener and sending outgoing and response messages back to the correct connection, the JIFServer also provides synchronous message correlation for outgoing synchronous messages. This means that when an outgoing Request message is sent, the server will block the calling thread until a Response message is received on the same connection, at which time the response is returned. The correlation occurs at the server level so that the listener only needs to manage the BCs and their session IDs, and the BCs simply need to make a call to their CommunicationHandler. Additional information follows.

BusinessComponentListenerImpl

This class is used by the message transport system to process incoming messages. The BusinessComponentListenerImpl implementation class handles a collection of BC objects, keeping track of their session IDs and providing them with CommunicationHandlers (see below). In addition, this class converts all received XML messages and events to Java objects (using the JavaXMLConverter utility class), and invokes the requested method on the appropriate BC using the converted Java object as its argument data. When a response is returned from a request, or a message or event is raised from a BC, the class converts the Java object to XML and passes it to the message transport system. These messages and events are then communicated to the message transport either as returned objects, or through the JifMsgHandler interface with which the listener was instantiated.

XML remote procedure call

The POSBC XML requests are built using a format known as an XML remote procedure call (RPC). The root element of an XML remote procedure call (RPC) is the requested method name, and its first and only child element should contain the argument data to be passed to the method. This format is consistent with the schema used to define the RPC. The argument XML is converted to a Java object based on the schema specification used to define it. The following is an example of an Initialize request to the POSBC. In this example, the request XML shown below is converted to an InitializeRequest object, and sent as the argument to the initialize method:

```
public InitializeResponse initialize(InitializeRequest request)
```

```
soeps~Session-Id=1|Message-Type=REQ|~
<?xml version="1.0" encoding="UTF-8"?>
<xs:Initialize xmlns:xs="http://bc.si.retail.ibm.com/POSBCSchema">
  <InitializeRequest>
    <TerminalNumber>350</TerminalNumber>
  </InitializeRequest>
</xs:Initialize>
```

The response is returned as a Java object, converted into XML with attached soeps headers, and appears as follows:

```
soeps~Session-Id=1|Message-Type=RESP|~
<?xml version="1.0" encoding="UTF-8"?>
<schema:InitializeResponse xmlns:schema="http://bc.si.retail.ibm.com/POSBCSchema">
  <InitializeResult>
    <RequestID>1</RequestID>
  </InitializeResult>
</schema:InitializeResponse>
```

XML validation

A configurable option allows you to validate all incoming XML messages against a specified schema before they are processed. Locate this option in the bclistener.properties file. See“Configurability.”

Invalid messages create error logs that specify the XML errors, and then are ignored. However, invalid schema XML is not ignored: A generic response is sent back to the client of the following form:

```
[..Included here are soeps headers..]
<?xml version="1.0" encoding="UTF-8"?>
<schema:ExceptionResponse
<xmlns:schema="http://bc.si.retail.ibm.com/POSBCSchema">
<ExceptionResult>
<Message>Invalid request received</Message>
<ErrorCode>INVALID_REQUEST</ErrorCode>
</ExceptionResult>
</schema:ExceptionResponse>
```

BusinessComponent

The BusinessComponent interface defines the common methods that business components must implement. Since business components can offer any service, the common functionality is the ability to send messages and events. This function is accomplished through a CommunicationHandler, which is set on all business components when they are created through this common interface.

The CommunicationHandler interface offers the following methods:

- Object sendMessage(Object message) throws IOException
- void sendEvent(Object event) throws IOException

Also, BusinessComponents are provided with a bcConnectionDied() method that is called by the underlying messaging system when the connection to it has ended. This notification could then be used to trigger a shutdown, resource cleanup, state persistence, and others.

Configurability

The JIFServer and BusinessComponentListenerImpl classes provide configurable options that you can set using the appropriate properties files.

JIFServer

The jifsrvc properties bundle can be extended by the user through the standard SI bundling, such as in a userjifsrvc.properties file.

Table 20. Extending jifsrvc properties bundle

Property	Description
bclistener.class	BusinessComponentListener implementation class (com.ibm.retail.bc.BCListenerImpl class is provided as a default)
server.port	Port the server monitors for BC connections on (default 6679)
max.buffer.length	Largest acceptable message length, in bytes (default 262144 bytes)
event.logging	Set to true to log outgoing events to the configured file (default is false)
event.file	File where the outgoing events are logged; if empty, the events are logged to stdout. **Note:** If file path contains spaces, use %20 in the space.

BusinessComponentListenerImpl:
You can extend the bclistener properties bundle by using the standard SI bundling method, such as in a userbclistener.properties file.

Table 21. userbclistener.properties file

Property	Description
bc.class	BusinessComponent implementation class to construct when a session is requested (com.ibm.retail.si.bc.POSBCImpl class is provided as a default)
schema.validation	Set to true to validate incoming XML against configured schema (default is false)
schema.mapping	Defines the mapping of namespaces to its schema file location. Each set contains the URI of the namespace to be validated, paired with the absolute file path of the schema file that defines the namespace. Format in the following way: [(namespace 1, schema location 1),(namespace 2, schema location 2),...] **Note:** If file path contains spaces, use %20 in the space.

Interaction diagrams

Incoming message/event

In Figure 26 on page 67, the Client sends the BusinessComponent (an instance of POSBC) an InitializeRequest, which is the first message request. On receipt of this first request, a BusinessComponent is created for the Session-Id. In subsequent requests, the BusinessComponent is reused and referenced by the Session-Id. This same example is identical to an incoming event, except that the flow stops with a void return from the requested method on the BusinessComponent, and the method called on the BCListener would be onEvent instead of onMessage.

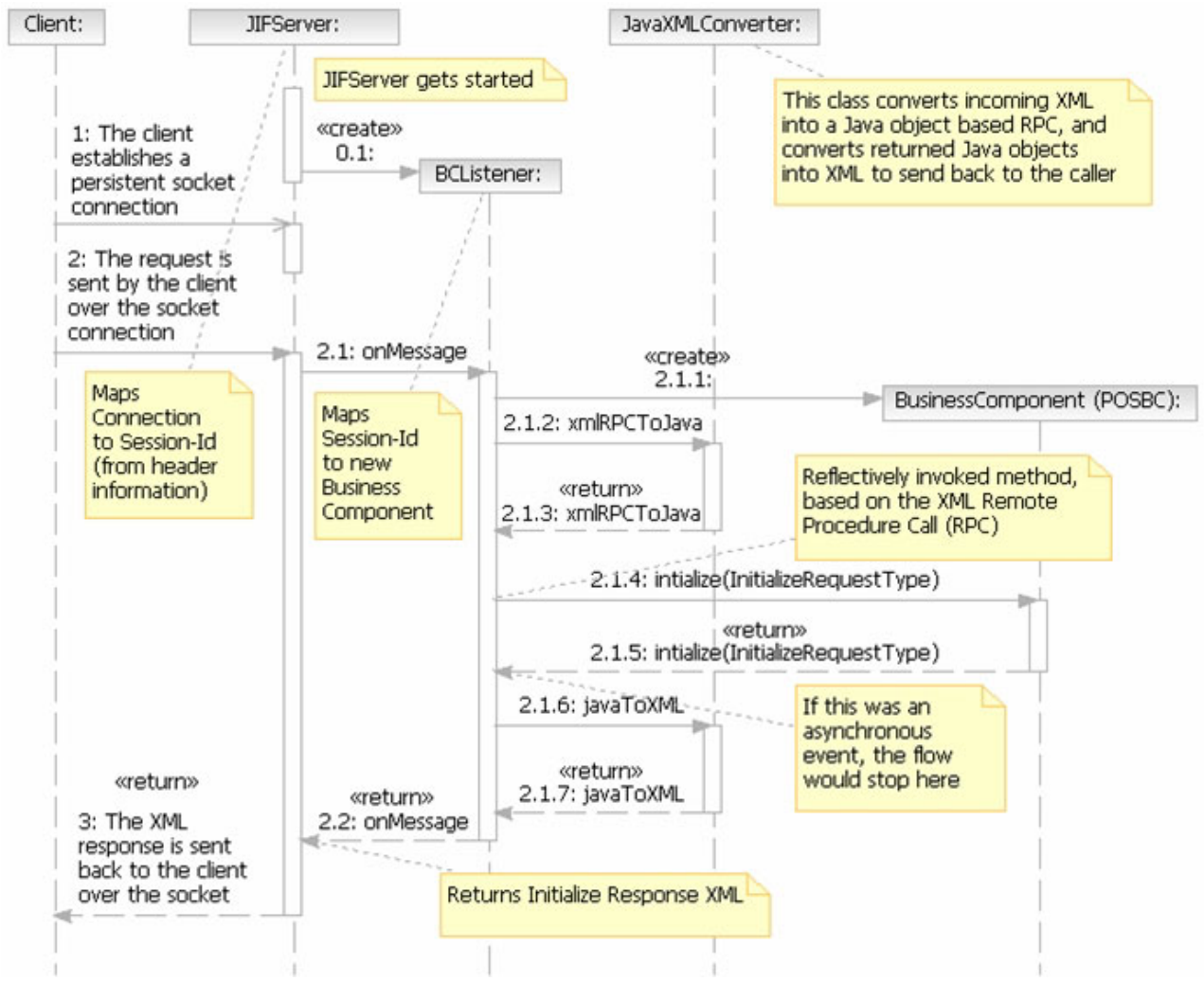
Client:
JIFServer:
JavaXMLConverter:
JIFServer gets started
This class converts incoming XML into a Java object based RPC, and converts returned Java objects into XML to send back to the caller
1: The client establishes a persistent socket connection
«create» 0.1:
BCListener:
2: The request is sent by the client over the socket connection
2.1: onMessage
«create» 2.1.1:
BusinessComponent (POSBC):
Maps Connection to Session-Id (from header information)
Maps Session-Id to new Business Component
2.1.2: xmlRPCToJava
«return» 2.1.3: xmlRPCToJava
Reflectively invoked method, based on the XML Remote Procedure Call (RPC)
2.1.4: intialize(InitializeRequestType)
«return» 2.1.5: intialize(InitializeRequestType)
2.1.6: javaToXML
If this was an asynchronous event, the flow would stop here
«return» 2.1.7: javaToXML
«return» 2.2: onMessage
«return» 3: The XML response is sent back to the client over the socket
Returns Initialize Response XML

Figure 26. Message flow.

1. The client establishes a persistent socket connection.
2. The request is sent by the client over the socket connection.
3. The XML response is sent back to the client over the socket.

Outgoing message/event

In Figure 27 on page 68, the BusinessComponent sends a message request to the client using the CommunicationHandler that initialized the request. The CommunicationHandler converts the request to XML, and then passes the message to JifMsgHandler, which the BCListener initialized. The message is sent to the Client, and execution is blocked until the Client responds. The response message then converts to a Java object and is returned to the BusinessComponent. This same example is identical to an outgoing event, except that the flow stops after the event message is sent to the Client. Also, the sendEvent methods are called instead of sendMessage methods.

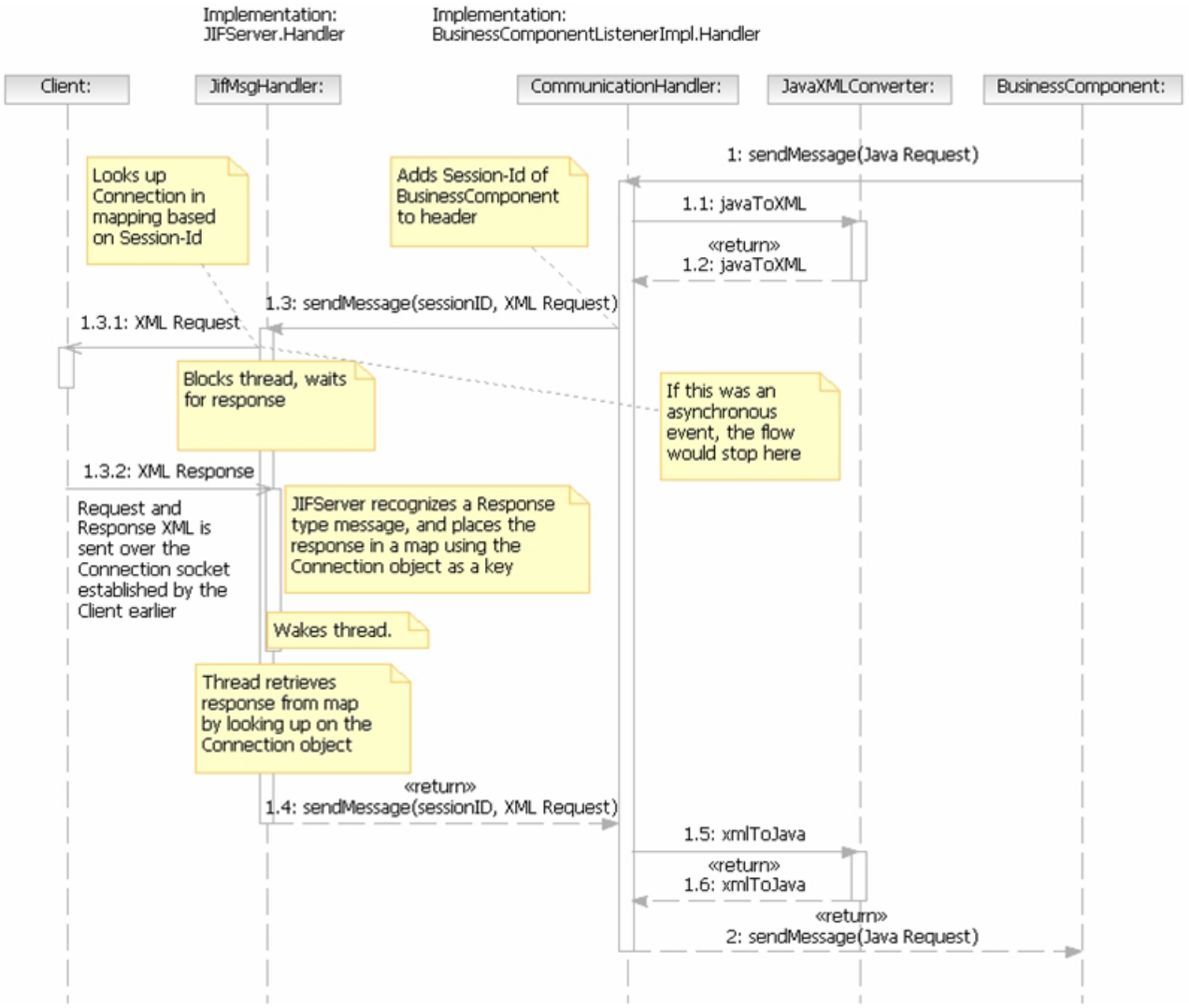

Figure 27. Message flow

POSBC main classes

This section describes the functions of several of the main classes in the POSBC API. For more detailed information on these classes and methods, see the JavaDoc.

POSBCImpl

This class handles all the requests to the POS application.

POSApplicationAEFImpl

This class is the AEF Implementation of the POSBC interface that is called by POSBCImpl. It interacts with the POS automation provider.

If the client terminates the POSBC, by sending a terminate request, this class will cease to send events. Event flow resumes once the POSBC is initialized again. Any events that occur while the POSBC is terminated will not be queued and sent upon reinitialization.

Virtual receipts

POSBC constructs virtual receipts based on the cash receipt lines received from the POS application. By using virtual receipts, you can add to the receipts, remove lines from the receipts, print the receipts more than once, and so on.

A POSReceipt maintains information on a current POS transaction in ordered sections:

- Header
- Body
- Trailer

Each of these sections is subdivided into groups. A single group consists of all relevant receipt information in the form of an ordered array of FormattedReceiptLineType objects associated with a single operation in a transaction.

Additions, updates, and removals to a POSReceipt might be triggered using the following sources:

- CashReceiptEvents sent from the POS application through the POSBC and
- client AddReceiptLinesRequest, RemoveReceiptLinesRequest, and VoidLastItemRequest calls to the POSBC.

With the creation of a new group, an update to an existing group, or a removal of an entire group, a POSReceiptEvent consisting of the update type, receipt type, section ID, group ID, and an array of all FormattedReceiptLineType objects for the group is broadcasted to listeners.

POSReceipts are accessible by transaction and organized first by a receipt Type and then by an Index under the corresponding Type. For instance, a Type of Customer and Index of 0 together indicate the first receipt of Type Customer in a transaction. Client calls to modify receipts in the current transaction are expected to use this structure.

Figure 28 illustrates the contents of POSReceiptEvents that clients receive:

```
<xsd:complexType name="POSReceiptEventType">
   <xsd:complexContent>
      <xsd:extension base="BasicEventType">
         <xsd:sequence>
            <xsd:element name="Type" type="xsd:string"/>
            <xsd:element name="Index" type="xsd:int"/>
            <xsd:element name="Section" type="xsd:string" minOccurs="0"/>
            <xsd:element name="Group" type="xsd:int" minOccurs="0"/>
            <xsd:element name="FormattedReceiptLineList" type="FormattedReceiptLineListType"
       minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
         <xsd:attribute name="UpdateType" type="xsd:string" use="required"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:element name="POSReceiptEvent" type="POSReceiptEventType"/>
```

Figure 28. XSD sample

Attributes

UpdateType

Reflects the type of update this event represents. The possible values are Add, Modify, Remove, and EndReceipt. An Add is received when lines are added as a result of a client request, such as AddItem. A Modify can be received for a previously sent Group. It will contain print lines for all previously sent lines in the Group, as well as any new lines. When receiving a Modify, you should discard any saved lines for the given Group and replace them all with the new POSReceiptEvent since it is a complete replacement. A Remove is received as a result of lines

being removed, such as a VoidLastItem request with the CleanVoidFlag set to true. An EndReceipt is received when the receipt is complete.

Elements

Type Indicates the type of the receipt associated with this event. Default types supported for each transaction include Customer, RegE, Gift, and Warranty. Additional types might be added through the AddReceiptLines operation.

Index Begins at 0 and corresponds to the index of the receipt under all receipts of the above Type in this transaction.

Section
Indicates the section of the receipt to which this event applies. Possible values are Header, Body, and Trailer.

Group Identifies the group of lines within the above Section that are affected by this event. If UpdateType is Add, Group will be a new, unique value. If UpdateType is Modify or Remove, Group will correspond to an existing value. If UpdateType is EndReceipt, the Section and Group values are omitted.

The XSD sample below illustrates the contents of the FormattedReceiptLines that clients receive in POSReceiptEvents:

```
<xsd:complexType name="FormattedReceiptLineType">
   <xsd:sequence>
      <xsd:element name="Displayable" type="xsd:boolean" default="true" minOccurs="0"/>
      <xsd:element name="Printable" type=:xsd:boolean" default="true" minOccurs="0"/>
      <xsd:element name="Feeds" type="PMLfeeds" default="1" minOccurs="0"/>
      <xsd:element name="Mapmode" type="PMLmapmode" default="dots" minOccurs="0"/>
      <xsd:element name="Align" type="PMLalign" default="left" minOccurs="0"/>
      <xsd:element name="LineCategory" type="xsd:string" minOccurs="0"/>
      <xsd:element name="LineType" type="xsd:string" minOccurs="0"/>
      <xsd:element name="TextReceiptLine" type="TextReceiptLineType" minOccurs="0"/>
      <xsd:element name="ImageReceiptLine" type="ImageReceiptLineType" minOccurs="0"/>
      <xsd:element name="BarcodeReceiptLine" type="BarcodeReceiptLineType" minOccurs="0"/>
      <xsd:element name="LogoReceiptLine" type="LogoReceiptLineType" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
```

Figure 29. POSReceiptEvents

Elements

Displayable
Indicates whether or not this line should be displayed.

Printable
Indicates whether or not this line should be printed.

Feeds Indicates the number of line feeds to advance the receipt after this line.

Mapmode
Specifies the unit used for printing and can be dots, twips, english or metric.

Align Can be left, center, right, or offset:x (n/a to text) where x is the distance from the left-most print column to the start of the barcode or image printed inside this alignment in units specified by Mapmode; valid values for x are between 0 and 99999 inclusive.

LineCategory
Identifies the type of line. The possible values are Header, TenderFrankingSecondCopy, NonSales, LineItem, Trailer, PartialReceipt, PostPrintReceipt, and GiftReceipt.

LineType
Identifies the type of line. The possible values are specified in POSBC_enums.xsd.

TextReceiptLine, ImageReceiptLine, BarcodeReceiptLine, LogoReceiptLine
The actual content of the receipt line is contained in one of these mutually exclusive tags.

Interaction examples

The following examples illustrate actions that can affect the POSReceipt and the corresponding POSReceiptEvents.

The POSBC receives the following CashReceiptEvents as a result of an AddItem request on an age-restricted item, *Red Wine*:

- The first event contains the receipt header lines that are a result of adding the first item to the transaction.
- The second event is the item information itself.
- The third event is a date-of-birth line added by the POS application.

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Header</Section>
    <Group>2</Group>
    <FormattedReceiptLineList>
     <FormattedReceiptLine>
   <Feeds>1</Feeds>
   <Align>center</Align>
   <LineCategory>Header</LineCategory>
   <LineType>StoreHeader</LineType>
   <TextReceiptLine>
    <Text>**************************************</Text>
   </TextReceiptLine>
     </FormattedReceiptLine>
     <FormattedReceiptLine>
   <Feeds>1</Feeds>
   <Align>center</Align>
   <LineCategory>Header</LineCategory>
   <LineType>StoreHeader</LineType>
   <TextReceiptLine>
    <Text>======= Welcome to SUREPOS ACE ======</Text>
   </TextReceiptLine>
     </FormattedReceiptLine>
     <FormattedReceiptLine>
   <Feeds>1</Feeds>
   <Align>center</Align>
   <LineCategory>Header</LineCategory>
   <LineType>StoreHeader</LineType>
   <TextReceiptLine>
    <Text>Your checker today is Operator 500 </Text>
   </TextReceiptLine>
     </FormattedReceiptLine>
     </FormattedReceiptLineList>
</POSReceiptEvent>
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>3</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>center</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>        Red Wine                7.99 B</Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
   </FormattedReceiptLineList>
```

Figure 30. Examples of POSReceipt and POSReceiptEvents (Part 1 of 2)

```
</POSReceiptEvent>
<POSReceiptEvent UpdateType="Modify">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>3</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>center</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>        Red Wine                7.99 B</Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>DateOfBirth</LineType>
            <TextReceiptLine>
                <Height>1</Height>
                <Width>1</Width>
                <Bold>0</Bold>
                <Text>Date of Birth = Sun May 30, 1982      </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
<POSReceiptEvent
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>3</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>center</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>        Red Wine                7.99 B</Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>DateOfBirth</LineType>
            <TextReceiptLine>
                <Height>1</Height>
                <Width>1</Width>
                <Bold>0</Bold>
                <Text>Date of Birth = Sun May 30, 1982      </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 31. Examples of POSReceipt and POSReceiptEvents (Part 2 of 2)

The client sends the following AddReceiptLinesRequest XML data to the POSBC for *Red Wine* since it is an age-restricted purchase.

```
<xsd2:AddReceiptLines xmlns:xsd2="http://bc.saf.retail.ibm.com/POSBC/schema/POSBCSchema">
<AddReceiptLinesRequest>
    <RequestID>3</RequestID>
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Header</Section>
    <OriginalRequest>
        <RequestID>3</RequestID>
        <Location>After</Location>
    </OriginalRequest>
    <FormattedReceiptLine>
        <Feeds>1</Feeds>
        <Align>1</Align>
        <LineCategory>LineItem</LineCategory>
        <LineType>AgeRestricted</LineType>
        <TextReceiptLine>
            <Height>1</Height>
            <Width>1</Width>
            <Bold>1</Bold>
            <Text>**Age Restricted Item**</Text>
        </TextReceiptLine>
    </FormattedReceiptLine>
</AddReceiptLinesRequest>
</xsd2:AddReceiptLines>
```

Figure 32. AddReceiptLinesRequest XML data

The following POSReceiptEvent is returned:

```
<POSReceiptEvent UpdateType="Modify">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>3</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>

     <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>        Red Wine   7.99 B</Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>DateOfBirth</LineType>
            <TextReceiptLine>
                <Height>1</Height>
                <Width>1</Width>
                <Bold>0</Bold>
                <Text>Date of Birth = Sun May 30, 1982 </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>AgeRestricted</LineType>
            <TextReceiptLine>
                <Height>1</Height>
                <Width>1</Width>
                <Bold>1</Bold>
                <Text>**Age Restricted Item** </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 33. Returned POSReceiptEvent

The POSBC receives two CashReceiptEvents as a result of an AddItem request on “Dog Food” with a linked discount.

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>4</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text> Dog Food     19.99 B </Text></TextReceiptLine>
       </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
<POSReceiptEvent UpdateType="Modify">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>4</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
<LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text> Dog Food      19.99 B </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
<LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text> Discount 1.5   1.50-B </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 34. CashReceiptEvents as a result of AddItem

The POSBC receives one CashReceiptEvent as a result of an AddItem request on “Bagels.”

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>5</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
<LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>    Bagels      3.99 B </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 35. CashReceiptEvent as a result of AddItem

Bagels is voided; therefore, the POSBC receives a VoidLastItemRequest that removes the *Bagels* print line.

```
<POSReceiptEvent UpdateType="Remove">
   <Type>Customer</Type>
    <Index>0</Index>
   <Section>Body</Section>
   <Group>5</Group>
</POSReceiptEvent>
```

Figure 36. VoidLastItemRequest

Bagels is re-added, so POSBC receives one CashReceiptEvent as a result of the AddItem request.

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>6</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemSale</LineType>
            <TextReceiptLine>
                <Text>        Bagels                  3.99 B </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
     </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 37. CashReceiptEvent as a result of AddItem request

Due to a period of inactivity, the POSBC receives two CashReceiptEvents as a result of an automatic totaling from the POS application. The Group for events resulting from automatic actions from the POS application are set as the negative value of the previous Group.

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>-6<Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>TransactionTotal</LineType>
            <TextReceiptLine>
                <Text>         TAX                     1.86  </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
<POSReceiptEvent UpdateType="Modify">
    <Type>Customer</Type>
    <Section>Body</Section>
    <Group>-6</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>TransactionTotal</LineType>
            <TextReceiptLine>
                <Text>         TAX                     1.86  </Text>
            </TextReceiptLine>
      </FormattedReceiptLine>
      <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>TransactionTotal</LineType>
            <TextReceiptLine>
                <Text>    **** BALANCE                28.25  </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 38. CashReceiptEvents as a result of automatic totaling

The customer then decides not to purchase *Red Wine*. The POSBC receives one CashReceiptEvent as a result of a VoidItem request on that item.

```
<POSReceiptEvent UpdateType="Add">
    <Type>Customer</Type>
    <Index>0</Index>
    <Section>Body</Section>
    <Group>7</Group>
    <FormattedReceiptLineList>
        <FormattedReceiptLine>
            <Feeds>1</Feeds>
            <Align>1</Align>
            <LineCategory>LineItem</LineCategory>
            <LineType>ItemCancel</LineType>
<TextReceiptLine>
                <Text> CL MO   Red Wine                7.99-B </Text>
            </TextReceiptLine>
        </FormattedReceiptLine>
    </FormattedReceiptLineList>
</POSReceiptEvent>
```

Figure 39. CashReceiptEvent as a result of VoidItem

POSBC configuration

The following items are specified in a standard Java properties file, posbc.properties.

Table 22. Standard Java properties file: posbc.properties

Property	Description	Default
posbc.event.listeners	A comma-delimited list of events to listen for. The possible values for this property are: POSReceiptEvent, TotalsEvent, TransactionStatusEvent, POSBCStatusEvent, TillExchangeOccurredEvent, StoreOptionsEvent, DepartmentOptionsEvent, TenderOptionsEvent, TareOptionsEvent, ReasonListsOptionsEvent, ItemDiscountOptionsEvent, TransactionDiscountOptionsEvent, NoTaxOptionsEvent, LocaleOptionsEvent, CustomerEvent, FuelOptionsEvent **Note:** The user must define all of the events they wish to see when overriding this property (including the default events).	TotalsEvent, POSReceiptEvent, POSBCStatusEvent, TillExchangeOccurredEvent, TransactionStatusEvent, CustomerEvent, OptionsReloadNeededEvent, OptionsEvent, StoreOptionsEvent, DepartmentOptionsEvent, TenderOptionsEvent, TareOptionsEvent, ReasonListsOptionsEvent, TransactionDiscountOptionsEvent, NoTaxOptionsEvent, ItemDiscountOptionsEvent, FuelOptionsEvent
posbc.enabled.receipt	A comma-delimited list of receipt event-types the POSBC sends.	
posbc.enabled.receipt.linetypes	A comma-delimited list of receipt line types the POSBC sends to its client. If this property is blank, all line types are sent.	
posbc.api.history.list.size	Specifies the maximum number of POSBC API requests to retain.	10
posbc.initial.state	Specifies the default state of initialization. However, if a state is specified in the initialize request, it will override this value. The choices are SIGNED_OFF or SIGNED_ON	SIGNED_ON
posbc.start.device.server	This property instructs the POSBC to start or not start the device server for any instances of POSBC. The accepted values are "true" and "false"; setting to "false" indicates that no devices are used by any instance of POSBC.	true
posbc.switchback.enabled	Enable automatic switchback feature in the POSBC.	true
device.server.host aefio.device.group install.posbc.printer.hook	These properties are used to start an AEFSession using the specified device group.	 scojpos true
posbc.printer.config	The following parameter specifies the printer configuration. REMOTE expects that a printer is properly configured remotely and instructs the POSBC to query the DeviceServer for the printer control. NONE bypasses all printer configurations on systems that do not use one. Any other value will be treated as a local printer initialization request using the provided value as the name of the printer in the jpos.xml file.	REMOTE

Table 22. Standard Java properties file: posbc.properties (continued)

Property	Description	Default
posbc.printer.receipt.midtxn.header posbc.printer.receipt.midtxn.trailer	Mid-transaction receipt headers/trailers. Use
to indicate extra return lines.	~~MID-TRANSACTION RECEIPT~~
~~MID-TRANSACTION RECEIPT~~
posbc.printer.receipt.reprint.char	Reprint receipt replacement character for whitespace (single character only).	.
posbc.printer.receipt.feed.count posbc.printer.receipt.cut.percent	Receipt printing properties	5 95
posbc.receipt.memory.size	Number of transaction receipts to maintain in memory	4
posbc.printer.debug.output	Printer output debug file. For debugging purposes only, set to a full filename (such as posbc.printer.debug.output=C:/debug.txt)	
posbc.session.server.uri	Point to a specific SessionServer URI, if none is specified in InitializeRequest. Default value is blank to allow you to connect to any SessionServer that responds.	
ready.wait.timeout	Session creation timeout in milliseconds. This value controls the time to wait for session to be ready for input. A value of -1 will return the session without waiting for a ready event.	120000
enable.suspend.chit suspend.chit.header.text	Enable suspend chit. This enables/disables the short suspend receipt. This receipt only contains a configurable header, the tax, balance, suspend number, suspend barcode (if enabled), and workstation information.	false Take this Voucher to the Payment Station

Failover

If your primary controller fails, failover allows you to continue using your virtual sessions. When an interruption is detected, a session will fail over from the failing Primary controller to a specified Backup controller. Failover is supported only for ACE (both full and partial failover) and for SA (full failover only).

Solution overview

Failover types and the solutions for each scenario are shown below.

Table 23. Failover types and solutions

		Terminal Number Integrity	
		Yes	**No**
Transaction Integrity	**Yes**	Duplicate persistence files to a Backup controller to recover data on that controller in the event of a failure. Pre-load same-named terminals on the Backup controller to quickly start them when failing over.	Duplicate persistence files to a Backup controller to recover data on that controller in the event of a failure. Pre-load terminals for use in the event of a failover; if the same terminal number is not used then store personnel will need to perform a terminal transfer.
	No	Pre-load failover terminals on the Backup machine, complete the startup when needed.	No extra failover work to be done. Use any available session and clear it before use.

For the scenarios in the top row of the chart, the client cares about transactional integrity when failing over to a Backup controller. Since a solution is already provided for clients who also care about terminal number integrity when failing over, the same solution is provided to clients who do not care about terminal number integrity. This scenario is the most demanding of the solutions and is referred to as *full* failover. For clients who care about terminal number integrity, but not transactional integrity, the basic design for passing ownership of a session back and forth between controllers is used, but the duplication of persistence files is omitted. This omission reduces a possible performance degradation from the remote

file I/O when duplicating and synchronizing the files. This scheme provides *partial* failover. In the event that a transaction is in progress when the controller fails over, the client is responsible for replaying the transaction. Switchback is automatic in this case with the default POSBC implementation.

Note: As illustrated in Table 24, when using partial failover with a Remote GUI, the Remote GUI does not have any persistence to enable it to retain the transaction data. When a failover occurs the transaction cannot be automatically replayed.

Table 24 defines partial failovers.

Table 24. Partial failover scenarios

	Terminal number integrity	**Transaction integrity**
Remote GUI	Yes	No
Client persists transaction data for replay	Yes	Yes

Persistence files

Persistence files are necessary for transaction recovery and, internally, for failover. The NVRAM file and the Totals Save file contain the details of a single transaction in progress on a terminal session. These files are updated when aspects of the transaction change, such as adding an item or coupon. AEF writes the NVRAM files through a device hook. Totals Save file I/O directives (such as flush NVRAM to Totals Save, or clear Totals Save) are sent through the ACE (or SA) NVRAM hook and the Totals Save file is then modified by AEF. The internal persistence file holds the ownership stamp described in this section.

The failover code obtains its persistence file handles through the static class PersistenceFileManager. Obtaining a file handle requires a file key and the terminal number in use, which correspond to entries in the failover properties bundle. The references that get returned are cast as RandomAccessFiles, and the file I/O is performed through that API. Under the covers (for failover-enabled sessions), the code automatically duplicates necessary I/O operations on remote files (when the remote files are available), and disconnects from and reconnects to remote files appropriately.

Controller interactions

This section describes interaction scenarios.

Partial failover to a backup controller:
In this case, the Primary and Backup controllers have started up normally. An AEF client requests session 300, which is served by the Primary controller. After using the session for some time, a failure occurs rendering the Primary controller unable to serve session 300. At this time, the client detects that it can no longer access the session, and requests session 300 again by obtaining a SessionServer from its AEFBase instance, and calling getSession(300). The SessionServer recognizes that the Primary controller can no longer serve session 300 (through a test of its AEFSessionFactory) and attempts to use the Backup AEFSessionFactory to provide session 300. The Backup controller receives a request for the failover session and returns session 300 to the client. The client re-initializes the session, and replays any partial transactions that were in progress before the controller failure occurred.

Note: A terminal that was in transaction during failover might come up on the Backup controller in a secured state; however any regular request, such as addItem, will work. AEF will issue an automatic sign on and continue. In this case the default operator ID and passwords are being used. If the defaults are not being used, then a manual sign-on with the specified password must occur before continuing to process the transaction.

Full failover to a backup controller:
In this case, the Primary and Backup controllers have started up normally. An AEF client requests session 300, which is served by the Primary controller. After using the session for some time, a failure occurs rendering the Primary controller unable to serve 300. At this time, the client detects that it can no longer access the session, and requests session 300 again by obtaining a SessionServer from its AEFBase instance,

and calling getSession(300). The SessionServer recognizes that the Primary controller can no longer serve session 300 (through a test of its AEFSessionFactory) and attempts to use the Backup AEFSessionFactory to provide session 300. The decision to use the Backup controller is made by passing in the third party data of PRIMARY DUPLICATING into the session ownership matrix; see Table 26 on page 86. The Backup controller receives a request for the failover session, assumes ownership of session 300's persistence files, and begins serving that session.

Note: A terminal that was in transaction during failover might come up on the Backup controller in a secured state; however any regular request, such as addItem, will work. AEF will issue an automatic sign on and continue. In this case the default operator ID and passwords are being used. If the defaults are not being used, then a manual sign-on with the specified password must occur before continuing to process the transaction.

Switch back to primary controller:
After a session fails over to its Backup controller, the Primary controller eventually comes online and requests ownership of its session. The reason for returning to the normal setup as soon as possible is for load balancing purposes. The Primary controller sends a request to the Backup controller for control of session 300 through a normal multicast Factory Beacon. If full failover support is configured, then the Backup controller immediately synchronizes its persistence files with the Primary controller, and begins duplicating persistence file writing to it. Regardless of which failover support is configured, the Backup controller then notifies the AEFSession's client that a switch-back has been requested. After this notification, whenever the session running on the Backup is destroyed, its ownership will be transferred back to the Primary controller. If full failover support is configured, the transfer back to the Primary is accomplished by setting the ownership stamp of the Primary controller to `DUPLICATING`; see Table 26 on page 86. The AEF client will determine when to destroy the session, and the switch-back will be set after the destroy operation completes. In general, the best time to perform the switch-back is between transactions. Finally, the client will request session 300 again, which will now be served by the Primary controller, and continue using that session.

Note:

1. A terminal that was in transaction during failover might come up on the Backup controller in a secured state; however any regular request, such as addItem, will work. AEF will issue an automatic sign on and continue. In this case the default operator ID and passwords are being used. If the defaults are not being used, then a manual sign-on with the specified password must occur before continuing to process the transaction.
2. Switch-back notification does not occur immediately when CSS is restarted on the Primary controller. The factory beacons trigger the switch-back notifications, so the first notification is received only after the first factory beacon is sent from the Primary controller.
3. When using manual switch-back mode, the client application needs to be aware that a POS_SESSION_SWITCHBACK_NOT_AVAILABLE status event might be sent after a POS_SESSION_SWITCHBACK_AVAILABLE. This will occur when the primary has gone down again before switching back, and the client application will need to be prepared for this.

A Remote Controller fails during normal operation:
In this case, a session is being served on either its Primary or Backup controller, with full failover support configured. The other controller fails. This failure would not impede the operation of the session as long as the failed controller is not the Master File Server. However, persistence files will be out of sync on the failed controller. Until the remote hard drive is accessible again, input and output to the persistence files is written only to the local copy on the serving controller. When the other controller is online, it ensures that the remote copies of the session files are synchronized. The remote controller emits multicast Factory Beacons notifying the network when it is online; the serving controller uses these beacons as triggers to automatically synchronize its files with the remote controller. After the files are synchronized, the "writes" are duplicated to the remote hard drive.

In Figure 40, the serving controller is the Primary controller and the remote is the Backup, but the scenario is exactly the same if the roles are reversed.

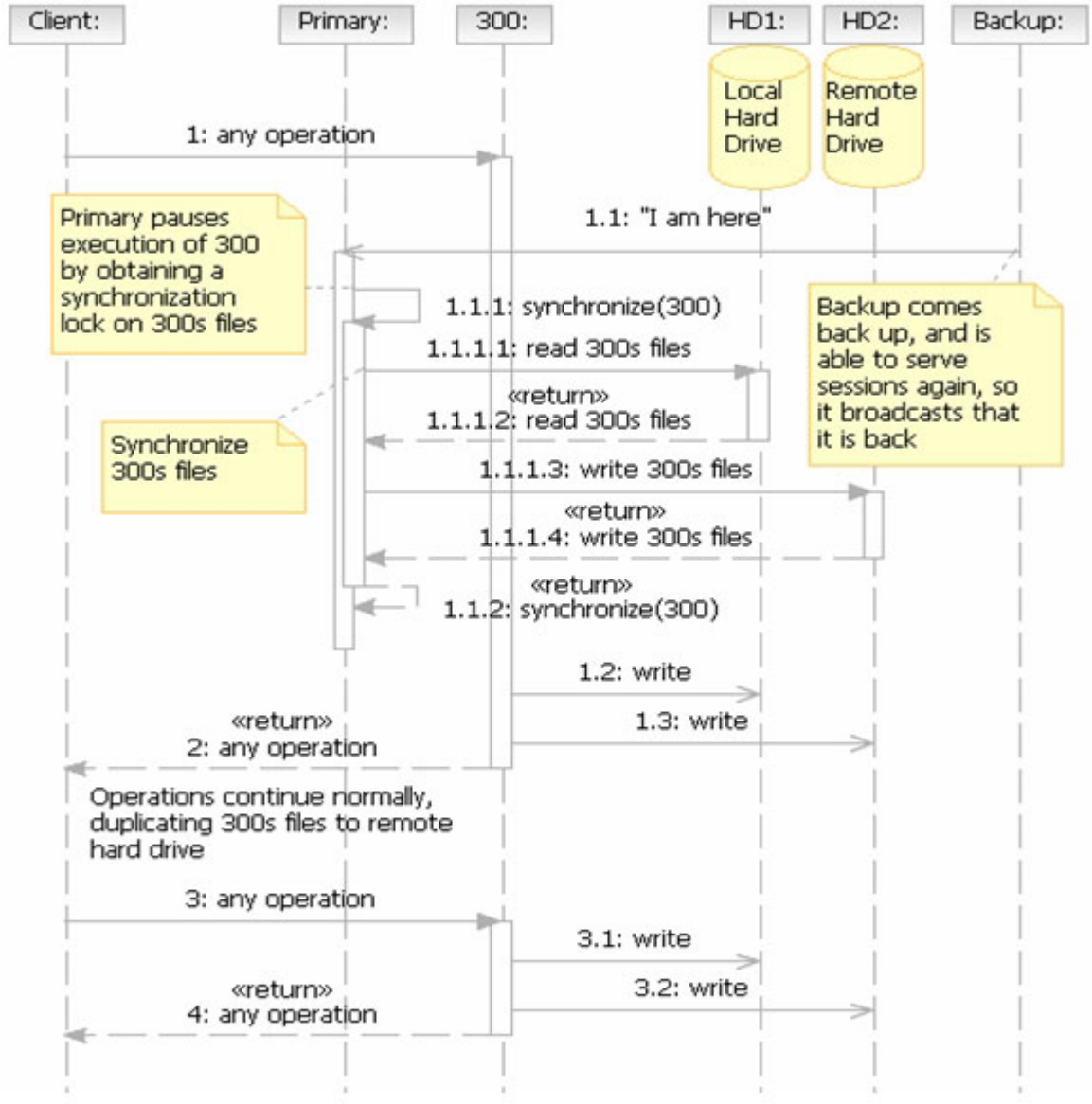

Figure 40. Diagram of backup controller fails during normal operation

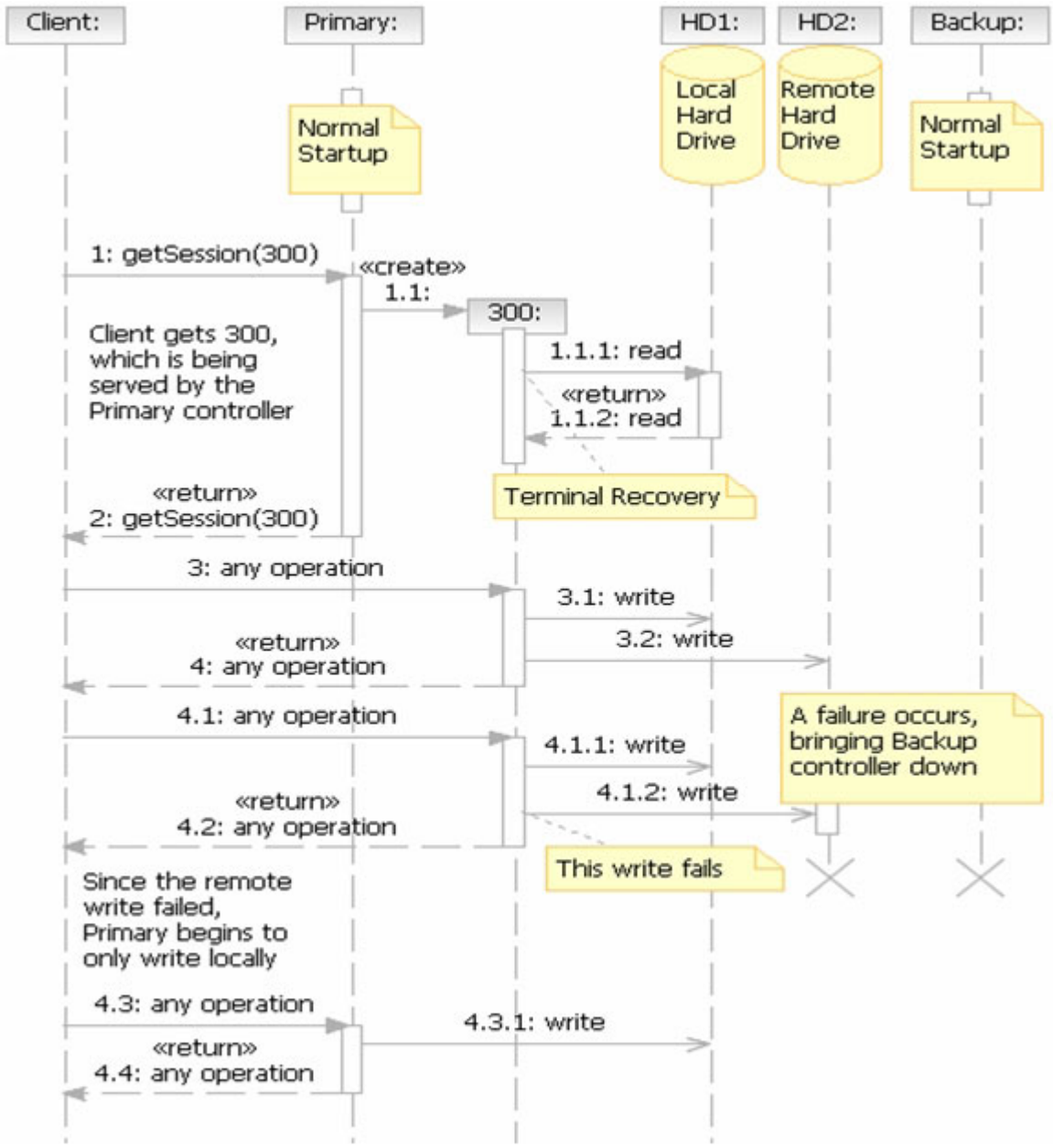

Figure 41. Diagram 2 of backup controller fails during normal operation

Messages

This section describes the messages used in failover.

Controller Switch Back notification:
The implementation of this notification is similar to other AEF listener interfaces: a client registers as a SessionOwnershipListener and then receives switchbackNotification() invocations. Notification of switchback will occur when the original owner of a session that was failed over is ready to take back ownership. When you receive this notification, you are responsible for destroying the session. After the event is received, the switch-back state is set when the session is destroyed. A typical client reaction to a switchbackNotification() message is as follows:

- Finish the current transaction and evaluate whether to pause and switch the session's controller (repeat as necessary).
- Block all future AEF interactions.
- Destroy the session in question through the SessionServer.
- Invalidate the AEFSession reference, get a new SessionServer, and query the SessionServer for a new session reference.
- Perform all typical cleanup that occurs when a session connection is lost (releasing proxy listeners, and other).
- After obtaining a new session reference, allow AEF interactions to continue. The new reference is from the Primary controller, having been switched back over as described previously.

Third party data change:
The third party data change notification and the controller switch-back notification share the same listener interface. A client registers as a SessionOwnershipListener and then receives thirdPartyDataChanged(SessionOwnershipEvent) invocations. Notification of a third party data change occurs when the file write mode of a session changes. Details of the third party data are explained in the following section.

Third party ownership voting

This section relates to full failover support. Partial failover support does not use third party data when determining session ownership. A loss of information occurs between two controllers when both go down and only one comes back up (or comes up first). In this scenario, a third party is needed to help determine which controller owns the persistence files. The AEF client program is the third party, and is responsible for maintaining a small piece of data that is provided to it each time the ownership of its session changes. The actual data is a String, either PRIMARY or BACKUP (indicating whether the Primary or Backup controller is in control at the time), followed by the actual ownership stamp of the session on that controller. The stamps are described in Table 25. When the AEF client requests a session (which triggers failover, and others) the third party data is passed in as a session parameter. If there is not enough information between the Primary and Backup controller to determine ownership, then the third party data is used. The client receives its third party data in the following ways:

- When first accessing a session, the data will be set on the session object, and should be read and stored (persisted, and others) before using the session.
- The client should then register itself as a SessionOwnershipListener on the AEFSession to receive changes to the data which it should always store. Table 26 on page 86 shows a matrix of how the data is used.

Note: If the AEF RecoveryModule class is used to maintain third party data, the stored file will be %SI_HOME%\user\RCMDATA.[terminal number]. The RecoveryModule is currently used by POSBC and Remote GUI for failover.

Ownership matrix

This section pertains to full failover support; partial failover support does not use the ownership matrix to determine ownership. A decision matrix determines whether the Primary controller or the Backup controller is given control of a session that is requested. The ownership stamps will be queried first, and should a conflict or missing data occur, then the third party data determines ownership. Table 25 describes the ownership stamps that controllers maintain for every session—both primary and failover sessions.

Table 25. Ownership stamps

Stamp	Meaning
BEING_DUPLICATED_TO	The controller is receiving duplicated data from the other controller.
DUPLICATING	The controller is duplicating its data to the other controller.

Table 25. Ownership stamps (continued)

Stamp	Meaning
LOCAL_ONLY	The controller is only writing data locally and is not duplicating data.

Table 26 is a matrix of the set of states of a Primary and Backup controller when determining the ownership of a requested session. The SessionServer attempts to retrieve the stamps from each of the controllers, and passes the stamps and the third party data provided in the session parameters through the logic defined in Table 26. In the absence of sufficient information, a session is not served since the state of the system is not known. In this case, an AEFException is returned with the error code SESSION_NOT_AVAILABLE.

Table 26. States of a primary and backup controller. To use this matrix, select the stamp of your Backup controller in the vertical column and follow the row until you match the stamp of your Primary controller stamp in the horizontal column.

	Primary Controller Stamp					
Backup Controller Stamp		NO STAMP	BEING DUPLICATED TO	DUPLICATING	LOCAL ONLY	DOWN
	No Stamp	Primary	Primary if 3rd[1] BACKUP DUPLICATING else wipe out files*	Primary if 3rd PRIMARY DUPLICATING else wipe out files*	Primary if 3rd PRIMARY [DUPLICATING | LOCAL-ONLY] else wipe out files*	No Recovery[2]
	BEING DUPLICATED TO	Backup if 3rd PRIMARY DUPLICATING else wipe out files*	Unexpected State**; wipe out files*	Primary	Primary	Backup if 3rd PRIMARY DUPLICATING else No Recovery
	DUPLICATING	Backup if 3rd BACKUP DUPLICATING else wipe out files*	Backup	Primary: This is the 'Switchback' state	Primary	Backup if 3rd BACKUP [DUPLICATING | LOCAL-ONLY] else No Recovery
	LOCAL ONLY	Backup if 3rd BACKUP [DUPLICATING | LOCAL-ONLY] else wipe out files*	Backup	Backup	Last known by 3rd else, No Recovery***	Backup if 3rd BACKUP [DUPLICATING | LOCAL-ONLY] else No Recovery
	Down	No Recovery	Primary if 3rd BACKUP DUPLICATING else No Recovery	Primary if 3rd PRIMARY BACKUP [DUPLICATING | LOCAL-ONLY] else No Recovery	Primary if 3rd PRIMARY [DUPLICATING | LOCAL-ONLY] else No Recovery	N/A

[1] *3rd* means that the third party (client) has '...' as its ownership data.

[2] No Recovery means that the session cannot be served at this time, and an AEF Exception is returned with the error code SESSION_NOT_AVAILABLE.

*If the files from one controller are deleted and the third party vote cannot confirm that the files still available are the latest, then the transaction held in the persistence data on both failover controllers must be cleared (deleted). AEF automatically handles the marking of any leftover persistence files (EAMTSxxx, NVRAMxxx) for deletion before starting the requested session. The session will then begin normally, however, without any previous transaction data.

Note: This scenario can only occur if you manually delete ownership stamp files, or if the stamps were never initialized.

**An `Unexpected State` can only happen if the files are tampered with, or in the unlikely event that a serving controller crashes in the exact instant in between a very particular pair of file writes: The Primary machine must be up but not running CSS, and the Backup machine must crash in between the file writes that transfer ownership to the Backup machine for failover.

***When both controllers have a stamp of `LOCAL-ONLY`, they each *vote* for themselves as the latest owner, and it is up to the third party (the client) to split the tie. If the third party data shows that the Primary controller was the last one in control, then it gets control, and vice-versa for the Backup controller. Without a third party vote, no way exists to determine which set of files were used last, and; therefore, there is no recovery. Manually delete the files to continue.

Note: Only in the extreme case described above (***) is manual intervention needed to regain access to a session. In all other cases, the *missing* controller simply needs to become available again.

POSBC transaction recovery

As a distributed environment, POSBC receives requests through a TCP/IP connection. The POSBC client can be located on another system in the network. Since the system consists of multiple interconnected machines, a need exists to anticipate and recover from failures of any individual machine, as well as network failures. Since the purpose of the POSBC is to provide support for processing point of sale transactions, it is critical that clients of the POSBC can determine the state of the system when failures occur, and take the steps necessary to recover from such failures.

Overview

The categories of the POSBC recovery support are as follows:

- Generate POSBC status events when the status of its connection to the 4690 controller changes.
- Return the following information in response to a QueryStatus request:
  - RequestID of the last successful request.
  - POSBC State (such as, IN_TRANSACTION, NOT_IN_TRANSACTION, INCONSISTENT_STATE, etc.)
  - An indication of whether the AEF session currently has an active transaction.
  - The transaction ID and totals if the AEF session has an active transaction.
  - The last several POSBC requests that were received. The number of saved requests is configurable through a property in posbc.properties. The default entry is shown below:

    ```
    # Specify the maximum number of POSBC API requests to keep.
    posbc.api.history.list.size=10
    ```

  - SI version information

System configuration specifics::

1. Configure your system for full NVRAM backup.
2. If the acting master/acting file server is down for an extended period of time during which the acting master/file server is required, activate the alternate master/file server as the acting master/file server (for instructions on how to do this see the *4690 OS Planning, Installation, and Configuration Guide* GC30-4133.) This will enable transactions to be retrieved.

3. ACE Fuel specific: Set ACE terminal options for pumps to allow fuel sales to be voided (**Misc | Fuel Interface | General | "Allow fuel sales to be voided or refunded"**). This setting will allow the POSBC client code at the pump to void a transaction containing a fuel item, which might be needed for recovery.
4. Support: Possible actions to recover from failure scenarios:
   a. Restart the POSBC client JVM on Windows if it terminates. If the POSBC command window is no longer open, or if the POSBC client is not able to connect to the POSBC socket, it is likely that the POSBC client JVM has terminated.
   b. Restart CSS on the 4690 controller if it goes down. This condition can be determined by issuing ADXCSS1L STAT on a 4690 command line. A persistent "Session not available" response from the POSBC is also an indication that CSS on the 4690 controller may be inactive.

Client application responsibilities:
The client responsibilities are described in more detail in the Use Case section. The following are general items to take into account:

- The client should use the POSBC_STATUS events that relay connection status information to know when the POSBC is able to receive and process commands related to an AEF session. If the POSBC_STATUS indicates that there is no valid connection to the POS controller, the client should take appropriate action to prohibit activity and provide appropriate guidance to the end user.
- If a remote exception result is received in response to a request, issue a "QueryStatus" request to determine the state of the POSBC and the underlying AEF virtual session.
- If the client needs to terminate in a way that allows the current state to be recovered, the client should send a "Terminate" request with the "VoidTransaction" element set to "false". This will prevent the POSBC from cleaning up the AEF virtual session and will cause the POSBC to persist its own state information. The POSBC will continue to persist any changes to its state until another request is received from the client.

  When the client needs to reconnect to the POSBC, it should send an "Initialize" request with the "Recovery" element set to "true". At this point, it is appropriate to issue a "QueryStatus" request to determine if the state of the POSBC has changed with respect to the client.

  This scenario can occur if the POSBC client determines that a pending shut down is imminent (such as, if the client is aware of power interruptions and is able to gracefully shut down prior to losing power).
- If the client breaks the connection to the POSBC, the POSBC will detect this and react as if it had received a "Terminate" request with "VoidTransaction" set to false.
- If the POSBC reports status that is inconsistent with the state that the client expects, the recommended action would be any of the following (listed in order of preference):
  – Suspend the transaction.
  – Void the transaction and replay it using the POSBC client-maintained persistent data. Note that pre-authorization requests do not need to be replayed. This action will enable the client application to recover from all inconsistent state scenarios.
  – Send an "Initialize" request to the POSBC with the "Recovery" element set to "false".

  Note: This causes any transaction in progress to be voided and a Signoff-Signon sequence to occur.

Failover-specific client application responsibilities:
These client responsibilities are only necessary when failover is enabled.

When an automatic switch-back starts, a POS_SESSION_SWITCHBACK_IN_PROGRESS status event is sent to the client. The client should treat this like a POS_CONNECTION_LOST event by preventing further requests until a 'connected' event is received. The client should listen for either a POS_CONNECTION_RESTORED event to continue requests, or a POS_SESSION_SWITCHBACK_CANCELLED event. If the switch-back is cancelled, due to an AEFException, the client should go back to using the last connection status they heard. For example, if the last connection status event heard was POS_CONNECTION_RESTORED, then requests can continue

again. On the other hand if the last connection event heard was POS_CONNECTION_LOST, then they'll have to keep waiting for a 'connected' event; CONNECTED_TO_POS, POS_CONNECTION_RESTORED, or POS_SESSION_READY. If, however, an attempt is made to process any POSBC request that involves the POS, then the request will be rejected with a return code of "SESSION_SWITCHBACK_IN_PROGRESS". This return code should be handled the same way as a POS_SESSION_SWITCHBACK_IN_PROGRESS status event. Remember, this is almost the same as processing a return code of "SESSION_NOT_AVAILABLE" when trying to use a session that is disconnected.

When automatic switchback is disabled, it is up to the client application using POSBC to perform the session destruction. The client application will receive a POS_SESSION_SWITCHBACK_AVAILABLE status event when switchback to the primary controller is now available. After receiving this message, the client application should terminate the session, preferably between transactions, and then create the session over again so that the session comes from the primary. Unlike the automatic switchback mode, the manual switchback will also have the possibility of sending a POS_SESSION_SWITCHBACK_NOT_AVAILABLE event. This is sent when the primary controller of the session goes down before the client application has performed the switchback. The client application should treat the manual switchback the same as when a automatic switchback and prevent any further requests to be sent to the session before the session is started on the primary controller.

Use cases

Background: The POSBC RecoveryModule is a daemon thread started by the POSBC on an Initialize request that monitors the AEFSession every several seconds to ensure that the connection to AEF is still up. This daemon thread is similar to the AEF HealthMonitor. When there is a change in the state of the connection, the RecoveryModule notifies its registered ConnectionListeners (which is usually just the POSBC that started it). After a connection is lost, the RecoveryModule actively attempts to re-establish the lost connection, and informs its listeners when it is successful. Therefore, all remote references are managed, at least indirectly, by the RecoveryModule.

This feature is described in the following use cases within the following context:

- The POSBC and POSBC client have been successfully started.
- A connection to a remote virtual session has been established.
- A session has been successfully initialized.

User interaction flows:
The interaction flows for this feature are as follows:

Use Case: POSBC loses connection to AEF during idle time:
This feature has the following basic interaction flow:

1. During a lull in activity, the connection from the POSBC to its AEF session is lost.
2. The RecoveryModule pings the AEFSession and gets a RemoteException or times out. The timeout value is configurable through a property in config.properties. The default entry is shown below:

```
# Recovery Module properties
# recovery.module.interval
# This is the interval or wait time between session status checks
# recovery.module.timeout
# This controls the time the recovery module waits before timing out waiting
# for the session to respond with its ready status. This value is specified
# in miliseconds.
recovery.module.interval=5000
recovery.module.timeout=15000
```

Figure 42. Default timeout value

3. The RecoveryModule informs POSBC that it is DISCONNECTED through the ConnectionListener interface.

4. POSBC sends a POSBCStatusEvent to its client with Severity='INFO' and State='POS_CONNECTION_LOST'.
   - The POSBC denies all further requests involving AEF calls until the connection state goes back to CONNECTED, and the AEFSession is ready. The error code returned in the ExceptionResult is "SESSION_NOT_AVAILABLE".
5. The POSBC client halts further processing until it receives a POSBCStatusEvent with Severity='INFO' and State='POS_CONNECTION_RESTORED'.
6. The RecoveryModule attempts to reconnect using a SessionServer multicast every few seconds.
7. The RecoveryModule eventually reconnects.
8. The RecoveryModule then informs the POSBC it is CONNECTED through the ConnectionListener interface.
9. The POSBC refreshes its remote AEF references, and sends the POSBC client a POSBCStatusEvent with Severity='INFO' and State='POS_CONNECTION_RESTORED'.
   a. The POSBC will now allow future requests involving AEF calls.

Use Case: POSBC loses its connection to AEF during an 'Add' or 'Void' operation:
This Use Case assumes that the POSBC has just lost its connection to AEF in the middle of an 'Add' or 'Void' request from the POSBC client. This feature has the following basic interaction flow:

1. The remote call to AEF results in a RemoteException being thrown.
2. The POSBC sets its state to INCONSISTENT_STATE.
3. The POSBC informs the RecoveryModule that its connection has died, through the setDisconnected() method.
4. The following actions occur on separate threads in any order:
   - The POSBC returns an Exception result to the request with the RemoteException.
   - The RecoveryModule informs the POSBC it is DISCONNECTED through the ConnectionListener interface.
   - The POSBC denies all further 'Add' and 'Void' requests (other than VoidTransaction and SuspendTransaction) until the state of the POSBC is no longer INCONSISTENT_STATE.

     Note: The only way to depart from INCONSISTENT_STATE is for the client to send an Initialize request to the POSBC.
5. The POSBC client halts further processing until it receives a POSBCStatusEvent with Severity='INFO' and State='POS_CONNECTION_RESTORED'.
6. When the connection is restored, the POSBC client sends a QueryStatus request. Depending on the information received, the client will then either:
   a. Attempt to void the transaction and then re-initialize for the next transaction
   b. Suspend the transaction and then re-initialize for the next transaction (This action is not appropriate for fuel clients)
   c. Simply re-initialize for the next transaction if, for example, the 'add' was a tender that completed the transaction successfully.

Use Case: POSBC loses connection to POSBC client:
This use case assumes that the POSBC client application is hung, and the client application will then either force a socket disconnect or terminate regularly and close the socket. This feature has the following basic interaction flow:

1. The persistent socket connection to POSBC (through the JIFServer) is closed.
   a. If there is an outgoing request (such as, a Callback), the JIFServer 'cancels' the outgoing request by waking its execution thread (which at this point is waiting for a response to its request) and returning a "null" result back to the POSBC.
2. The JIFServer notifies the POSBC that the connection has died.

3. The POSBC saves its current state, and goes into a 'persist mode' where all further changes to critical variables are saved.
   a. The POSBC remains in 'persist mode' until a new connection to the POSBC is created and a request is issued.
4. POSBC attempts to cancel any pending cancelable requests
5. At this point, POSBC will wait for an Initialize request with Recovery set to true for a "warm start" or false for a "cold start".

Use Case: POSBC Client sends a Terminate request:
This use case occurs when the POSBC client application hangs and needs to be reset, or a power event is received from the UPS, and others. In these cases, the client application might have time to send a request to the POSBC instructing it to back up state information before it abends.

This feature has the following basic interaction flow:

1. The client application sends a Terminate request.
2. If the tag 'true' is present:
   a. The POSBC voids the current transaction with AEF, initializes the AEF state to Signed Off, and releases the AEF session
   b. The POSBC removes all of its AEF listeners, and stops the RecoveryModule from running
   c. The POSBC state is set to NOT_INITIALIZED, where it then waits for an Initialize request
3. The POSBC saves its current state, and goes into a 'persist mode' where all further changes to critical variables are saved.
   - The POSBC remains in 'persist mode' until a new request is issued.

Use Case: POSBC client recovers from a disconnected transaction ("Warm start" scenario):
If there is a question as to whether a transaction should be recovered or if it is known that there was a transaction in progress that was persisted, then the POSBC might be asked to perform a "warm start" after which it can be queried for information regarding the (possibly) recovered transaction. This feature has the following basic interaction flow:

1. The POSBC client sends an Initialize request with a 'true' tag.
2. If the POSBC had been shut down before this request (because of a power outage, a computer reset, or other) then it reads the persistence information from disk and populates its state variables. Otherwise the POSBC simply refreshes its listener support with AEF and returns to its caller.
3. The POSBC client sends a QueryStatus request. Depending on the information received, the client application will then either:
   a. Continue the transaction where it left off
   b. Attempt to void the transaction and then re-initialize for the next transaction
   c. Suspend the transaction and then re-initialize for the next transaction.

      Note: This action is not appropriate for fuel clients.
   d. Simply re-initialize for the next transaction, if, for example, the 'add' request was a tender that completed the transaction successfully

Use Case: User Interactions when running with Failover Enabled:

- The POS controller becomes inoperative and cannot be restarted during the idle time of POSBC or while adding an item or coupon.

  The Backup controller will take over the session. No action is required by the POSBC client application.
- The POS controller becomes inoperative and cannot be restarted during tendering.

  The Backup controller will take over the session. No action is required by the POSBC client application.
- The POS controller dumps:

The Backup controller will take over the session and the Primary controller will resume control of the session when it comes back up. No action is required by the POSBC client application.

- The POS controller is rebooted to apply maintenance to the system

  The Backup controller will take over the session and the Primary controller will resume control of the session when it comes back up. No action is required by the POSBC client application.

Use Case: Fuel solution considerations/implications:

- The POSBC will not be responsible for voiding any credit pre-authorizations. That is the responsibility of the POSBC client application.
- POSBC client application responsibilities:
  - Keep persistent data to be able to replay the sequence of POSBC requests if necessary.
  - Reverse a credit pre-authorization request if a response is not received from the POSBC. After connectivity has been re-established after being down, re-submit the pre-authorization request.
  - If the POSBC client breaks the connection with the POSBC, for example the client application dies, the client application should re-initialize itself with recovery set to 'true'. Then, it should query the status of the POSBC. If the state is inconsistent with its saved state, then the client application should void and replay the transaction that is in progress.
- Limitations:
  - The POSBC relies on AEF's ability to void transactions, if necessary, to initialize a session for use. If a void transaction request fails, the only recourse the client application has it to use terminal transfer on the kiosk GUI. Then the transferred transaction can be completed.

Chapter 3. Programming with the AEF

Getting started

This section describes the AEF programming tasks.

Setting up the development environment

To modify AEF configuration properties, or extend the AEF, you must set up a Java development environment. At a minimum, you need a Java Development Kit (JDK), a text editor, and some additional ZIP and JAR files. For the following instructions, the development platform is Windows-based; however, any platform with the appropriate JDK is acceptable.

Java Development Kit and JavaPOS requirements

The Java Development Kit (JDK) Version should be Version 1.4.2 SR10 or higher. The JDK can be acquired through the JavaPOS 1.9.5 or higher Software Development Kit (SDK) package.

Download and install the SDK version of JavaPOS 1.9.5, which is the required version for complete JavaPOS and JDK development capability. JavaPOS can be downloaded from the Toshiba Global Commerce Solutions Web site www.toshibacommerce.com/support

If you are compiling with JDK Version 1.5 or higher you must specify the compile target to be Java 1.4. This is set differently depending on where you are compiling your extensions.

Command Line: Pass the -target=1.4 -source=1.4 flags to the javac command

Build.xml(ant): specify target="1.4" source="1.4" in the parameters for the task

Eclipse Project: Set the Java Compiler Compliance level to 1.4 in the properties for your project.

Additional files:
To create applications that use Store Integrator functionality, the Java compiler needs access to Store Integrator and 4690 specific classes. Copy the following files into the c:/si/si_jars directory on your development system.

Note:

1. Several of the jar files from SI Version 2 of the programmers guide were combined in one jar file for SI Version 3. This combined file contains all of the contents of these jar files. The new file name is aefsys.jar.
2. You should migrate to the following setup when migrating from SI Version 2 to SI Version 3. The deprecated jar files are: aef.jar, aefio.jar, c_loging.jar, simgmt.jar, mx4j.jar, mx4j_rmt.jar, posprov.jar, cmcompat.jar, and sicompat.jar. The deprecated jar files are still shipped with SI Version 3. They will be completely removed in future releases.

- File from the 4690\COM directory on Store Integrator CD:
  - siutil.jar
- Files from 4690\AEF directory on Store Integrator CD:
  - aefsys.jar
- Files from C:\java\lib directory on the 4690 controller:
  - os4690.zip
- Files from C:\java directory on the 4690 controller:

  - jpos4690.zip
  - poss4690.zip
  - jpos14.jar
  - ibmjpos.jar
  - jattach.jar
  - tss.jar
  - log.jar
  - comm4690.jar
- Files from 4690\AEF*xxx* directory on Store Integrator CD:

  xxx is the development application you are using (SA, GSA, or ACE). You need aeface.jar or aefsa.jar or aefgsa.jar.

User hierarchy

The examples in this guide show a specific directory structure. You must create the directories in Table 27 on the development machine.

Table 27. Development machine directory hierarchy

Directory	Purpose
c:/si	Base Store Integrator development directory.
c:/si/si_jars	Contains JAR and ZIP files required for building Store Integrator extensions.
c:/si/user	Root directory for user Java source code.
c:/si/userjar	Root directory for user property file overrides and user Java class files.

Setting up Eclipse to develop Store Integrator extensions

It is recommended that developers use the Eclipse IDE to develop SI custom extensions. This section will describe the necessary steps to properly configure the Java SDK in Eclipse, how to create your Eclipse Project, how to configure the build path for your project, and how to build your project.

Before completing these steps you should have completed the “Setting up the development environment” on page 93 section above, including copying the additional files to c:/si/si_jars and extracting the SIprogrammerguide.zip file to c:/si.

Note: These instructions were created using Eclipse 3.5.1. The screen images shown may or may not match your screen if you are using a different version of Eclipse.

Configuring your Java SDK in Eclipse

- From the menu in Eclipse select **Window -> Preferences**.
- Expand the left tree browser to the **Java -> Installed JREs** pane.

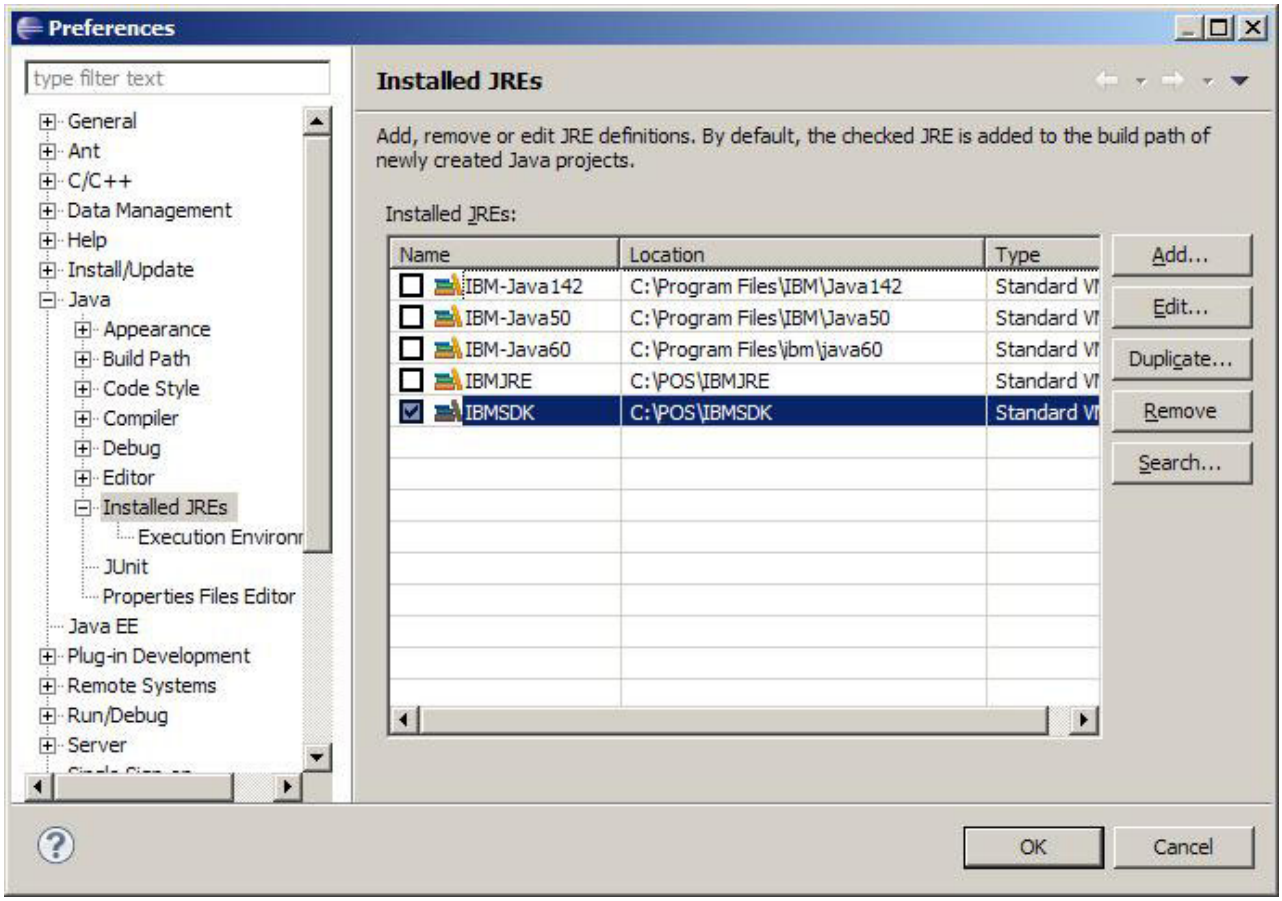

Figure 43. Configuring your Java SDK in Eclipse

- If Eclipse is not already configured with the C:\POS\IBMSDK Java SDK, then push the **Add**... button to add it.

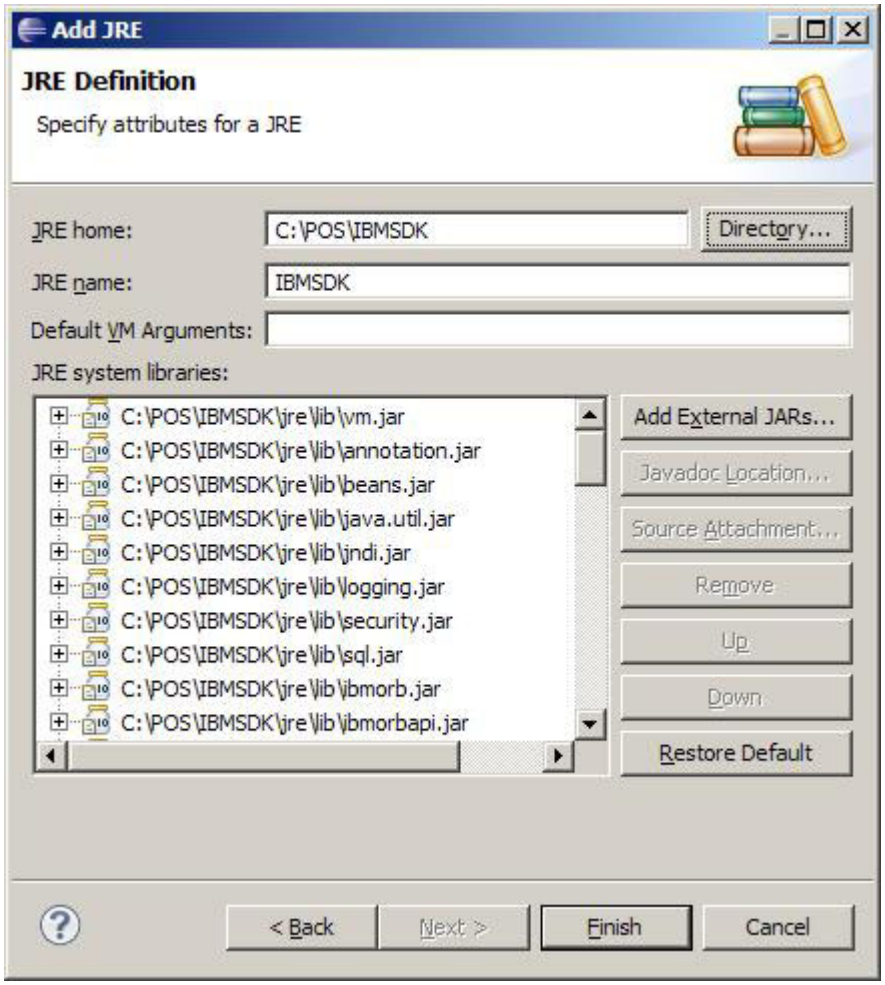

Figure 44. Add JRE

- Type C:\POS\IBMSDK into the **JRE home**: entry box. Once Eclipse loads the information about the SDK, click the **Finish** button.

Creating your Eclipse Project

- From the menu in Eclipse, select **File -> New -> Project.**
- Create your new project from the existing source in C:\si then click the **Next**> button.

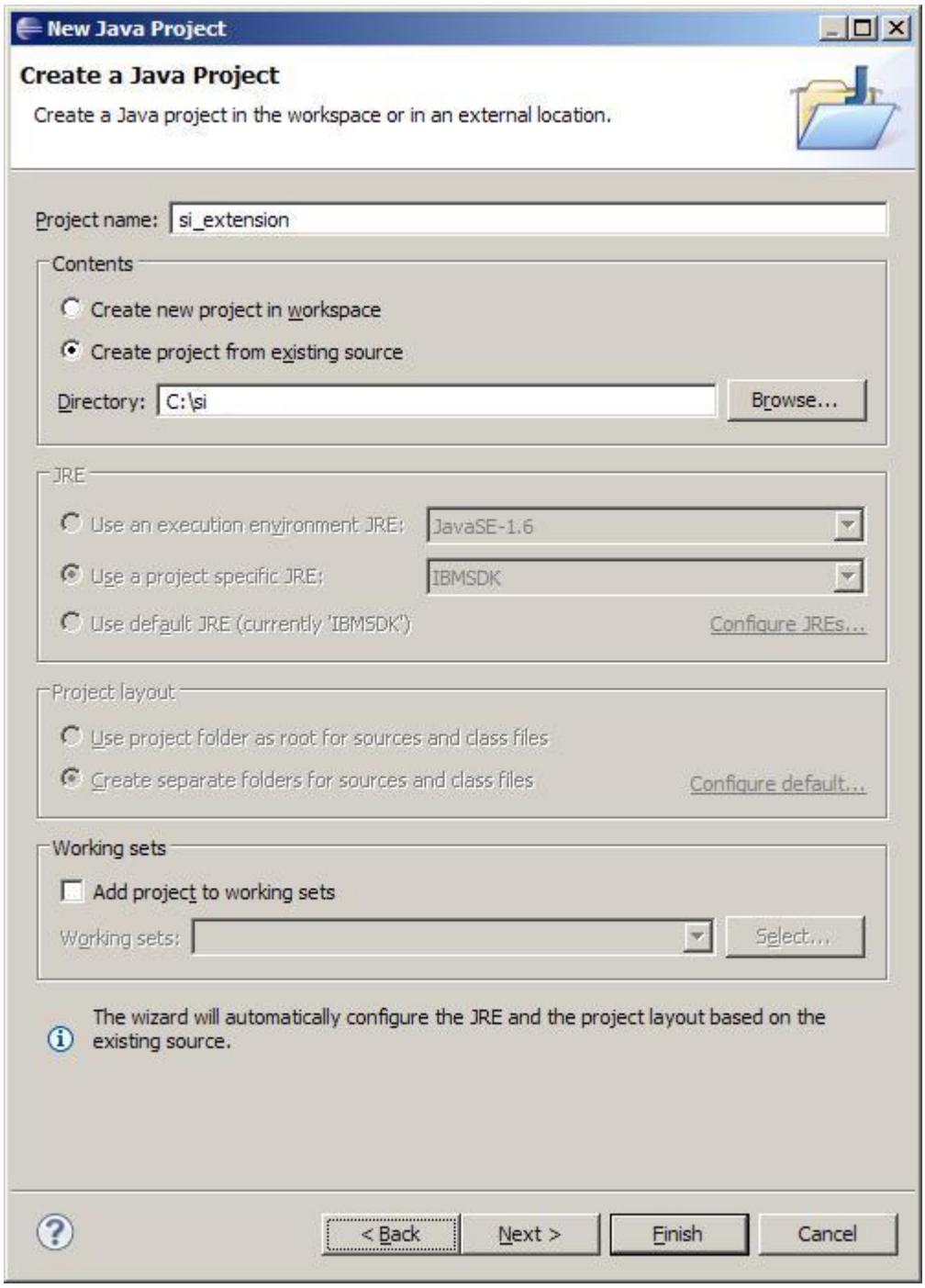
New Java Project
Create a Java Project
Create a Java project in the workspace or in an external location.
Project name: si_extension
Contents
Create new project in workspace
Create project from existing source
Directory: C:\si
Browse...
JRE
Use an execution environment JRE: JavaSE-1.6
Use a project specific JRE: IBMSDK
Use default JRE (currently 'IBMSDK')
Configure JREs...
Project layout
Use project folder as root for sources and class files
Create separate folders for sources and class files
Configure default...
Working sets
Add project to working sets
Working sets:
Select...
The wizard will automatically configure the JRE and the project layout based on the existing source.
< Back
Next >
Finish
Cancel

Figure 45. Creating your Eclipse Project

- Right click on the user folder and select the **Use as Source Folder** menu option.

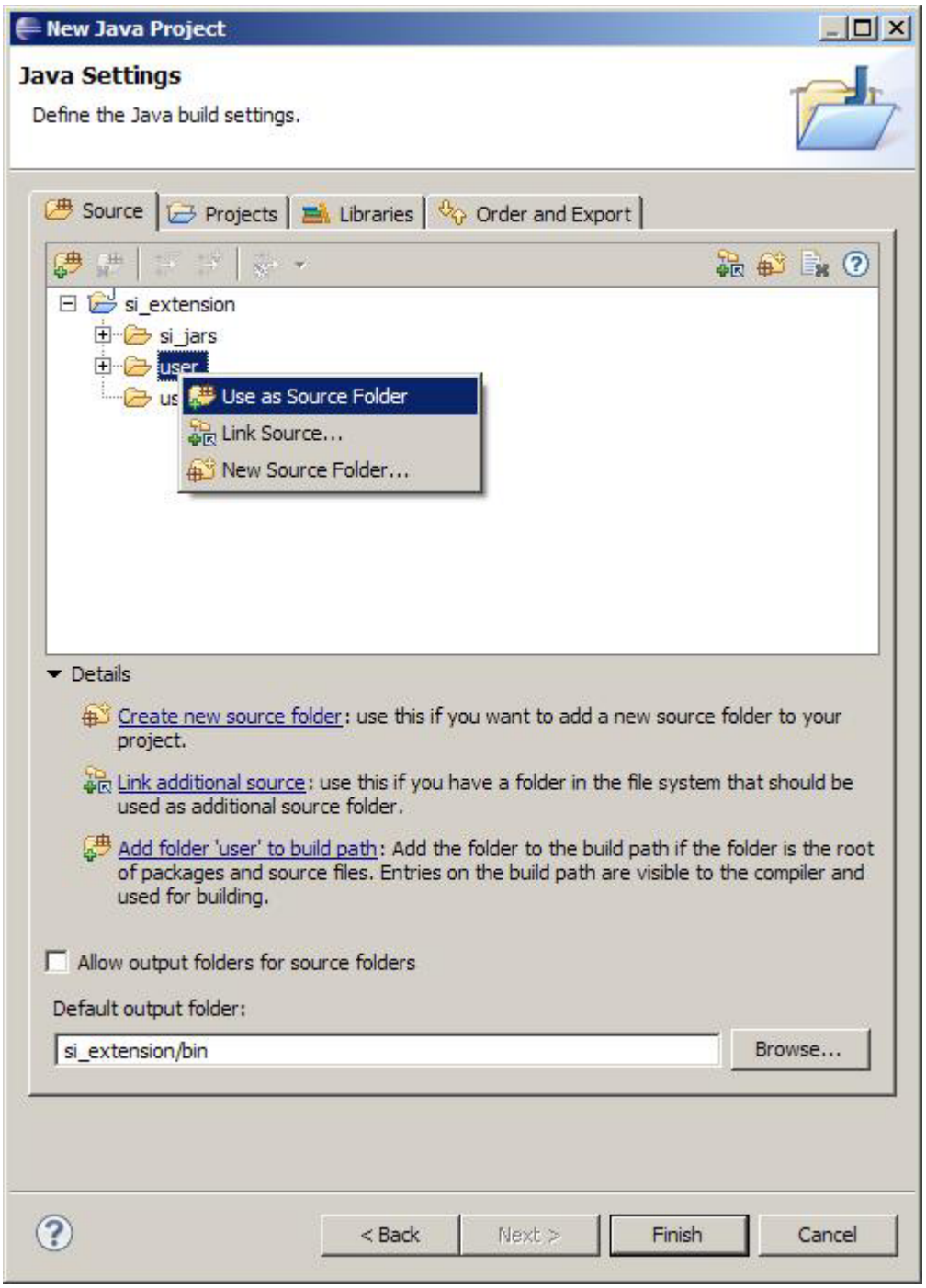
New Java Project
Java Settings
Define the Java build settings.
Source
Projects
Libraries
Order and Export
si_extension
si_jars
user
Use as Source Folder
Link Source...
New Source Folder...
Details
Create new source folder: use this if you want to add a new source folder to your project.
Link additional source: use this if you have a folder in the file system that should be used as additional source folder.
Add folder 'user' to build path: Add the folder to the build path if the folder is the root of packages and source files. Entries on the build path are visible to the compiler and used for building.
Allow output folders for source folders
Default output folder:
si_extension/bin
Browse...
< Back
Next >
Finish
Cancel

Figure 46. Java Settings

- Check the checkbox for **Allow output folders for source folders**.
- Set the **Default output folder:** to the user_jars folder in your project.

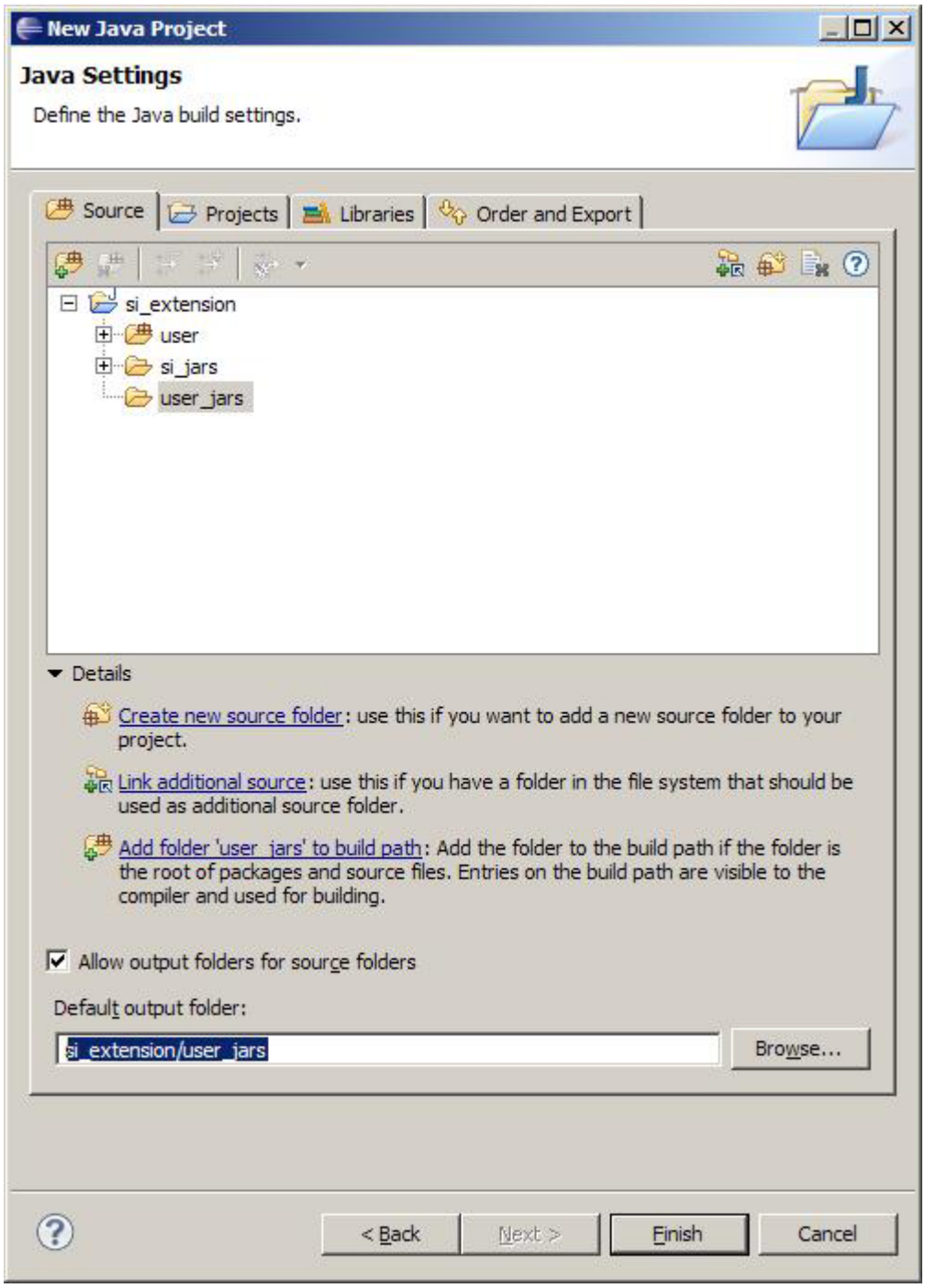
New Java Project
Java Settings
Define the Java build settings.
Source
Projects
Libraries
Order and Export
si_extension
user
si_jars
user_jars
Details
Create new source folder: use this if you want to add a new source folder to your project.
Link additional source: use this if you have a folder in the file system that should be used as additional source folder.
Add folder 'user_jars' to build path: Add the folder to the build path if the folder is the root of packages and source files. Entries on the build path are visible to the compiler and used for building.
Allow output folders for source folders
Default output folder:
si_extension/user_jars
Browse...
< Back
Next >
Finish
Cancel

Figure 47. Java Settings Default output folder

- Click the **Finish** button.

Configuring the Build Path for your Eclipse Project

- From the si_jars folder in your project select those files that are not already in the **Referenced Libraries** for the project. Right click on one of the selected files and select the **Build Path -> Add to Build Path** menu option. This will add those files to the **Referenced Libraries** for the project.

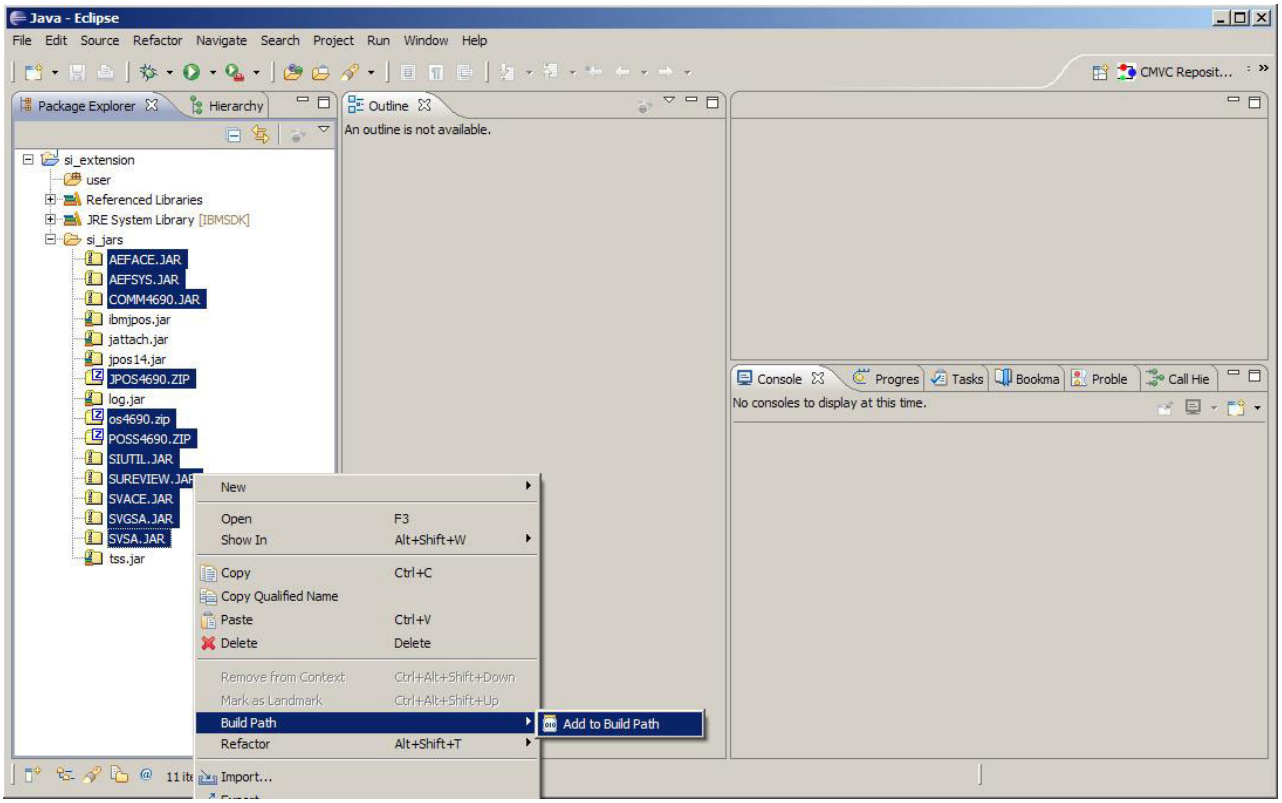

Figure 48. Configuring the Build Path

Setting the JDK Compliance level for your project to Java 1.4

- Right click on your project and select the **Build Path -> Configure Build Path** menu option.

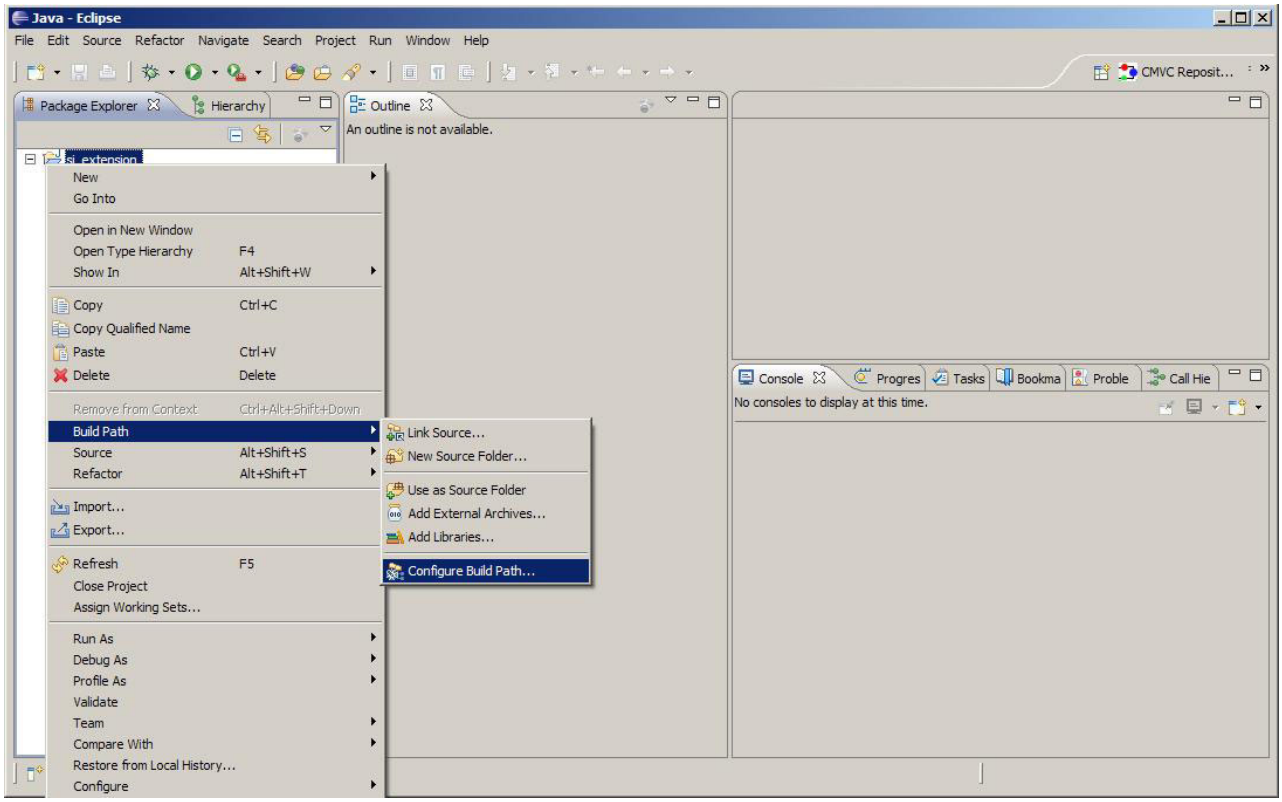

Figure 49. Setting JDK Compliance

- Select **Java Compiler** from the left tree browser.
- Set the **Compiler compliance level**: to 1.4
- Click the **OK** button.

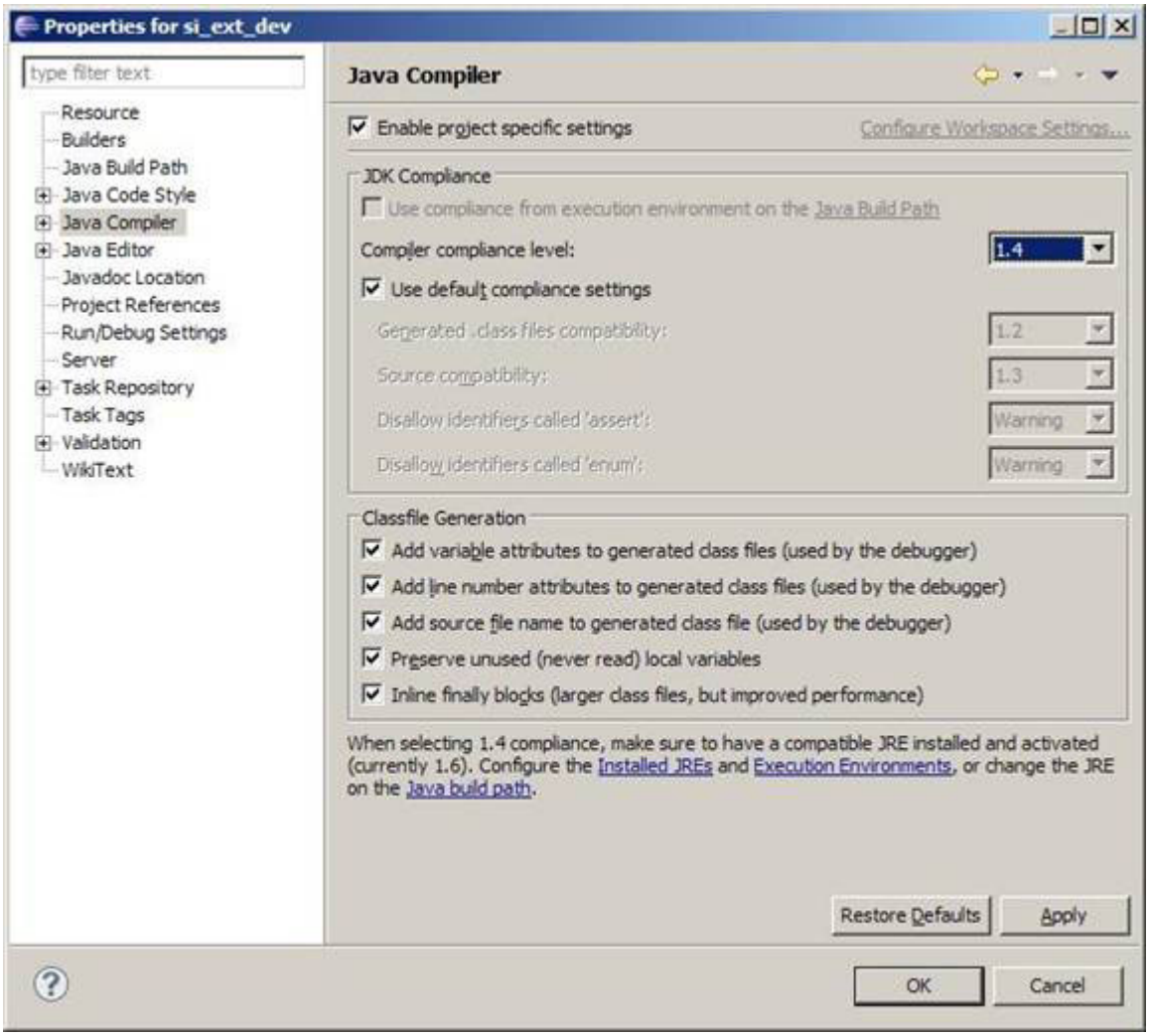

Figure 50. Compiler compliance level

Building (Exporting) your Eclipse Project

The project in eclipse should automatically rebuild as you make changes. This section describes how to export your project to a .jar file.

- Right click on your project and select the **Export...** menu option.

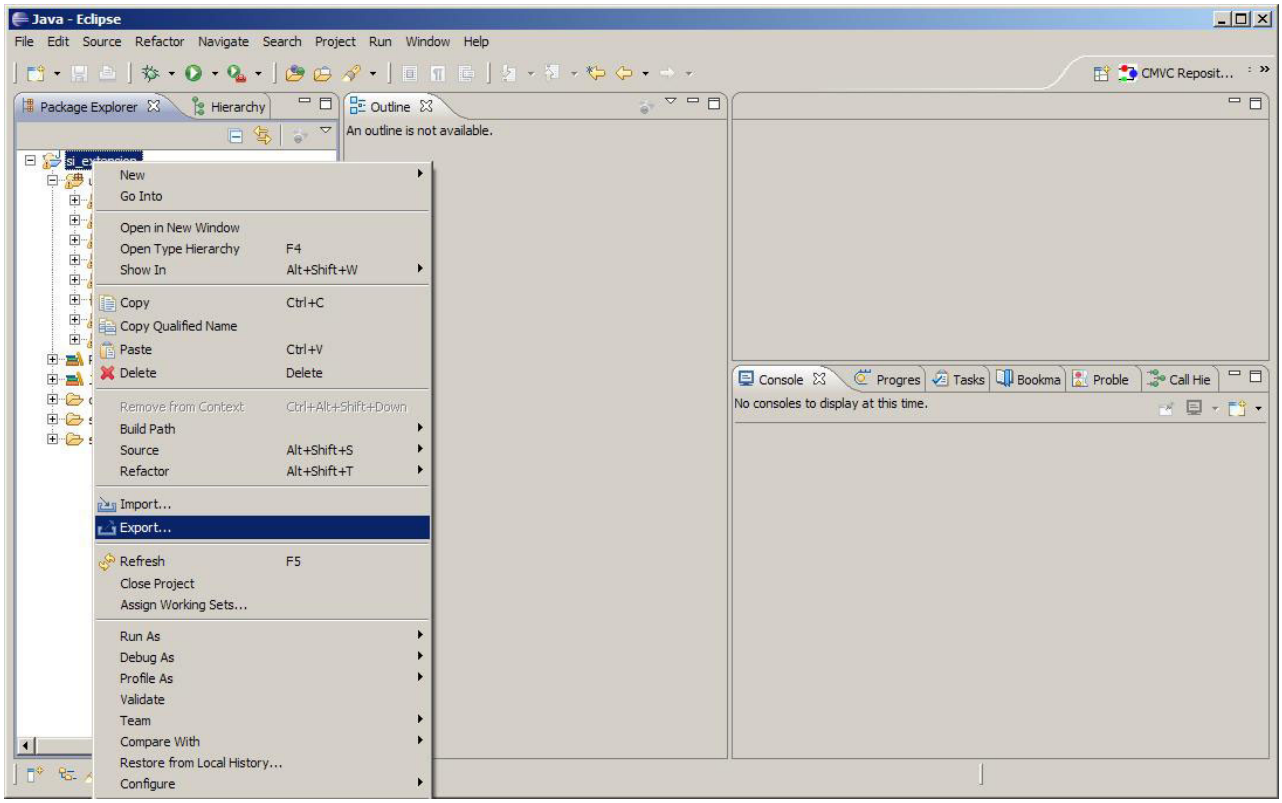

Figure 51. Exporting your eclipse project

- Select the **Java -> JAR file** option from the list and click the **Next>** button.

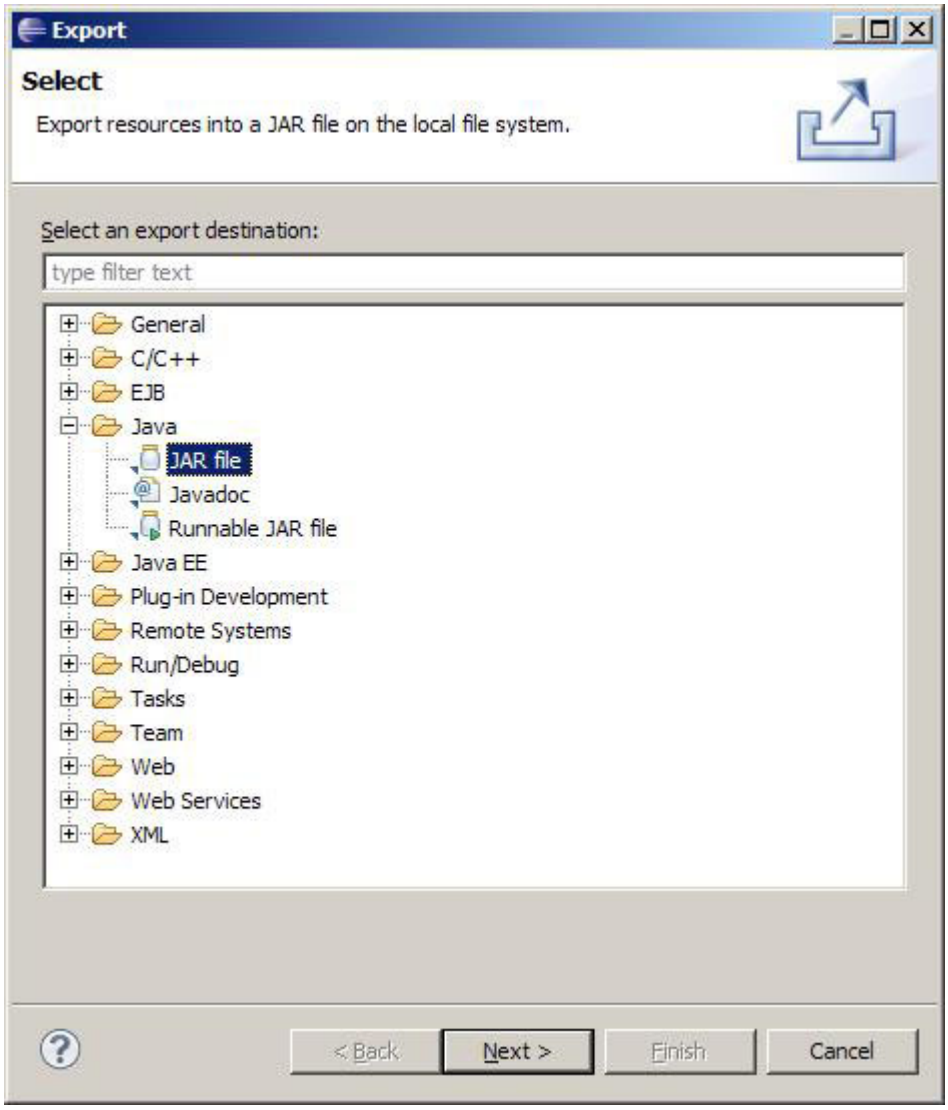

Figure 52. Exporting resources

- Select only the user folder from the **Select the resources to export**: boxes.
- Check only the checkbox for **Export all output folders for checked projects**.
- In the **JAR File**: entry box input si_extension/user_jars/siuser.jar to export to the siuser.jar file in your user_jars folder for your project.
- Click the **Finish** button.

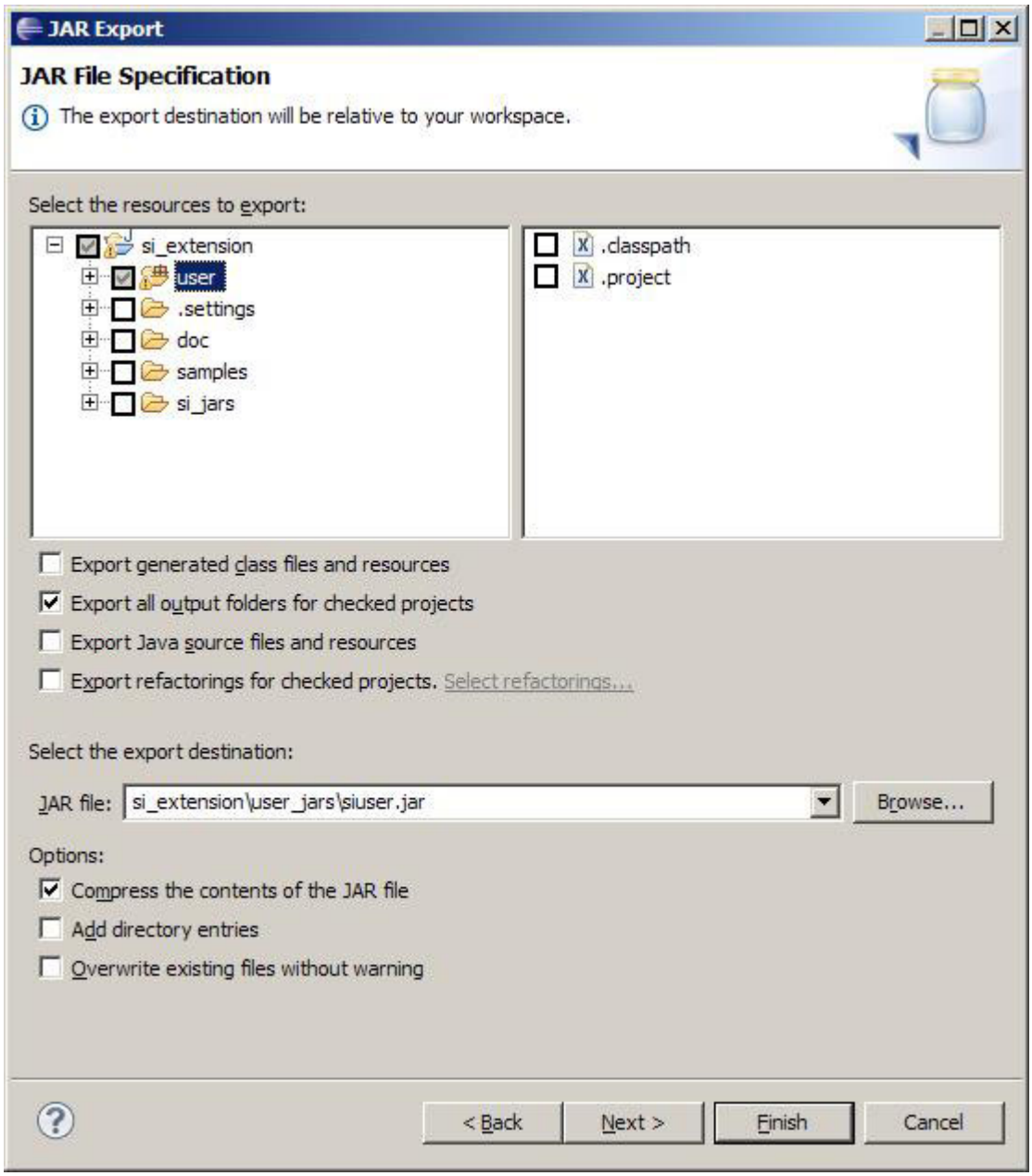

Figure 53. Exporting to si_user.jar

Using the quick start sample

This document (see "How to use this guide" on page xiii) is packaged with precompiled source for a sample AEF client program that demonstrates various aspects of the AEF API. You must have the following prerequisites to run the sample client.

1. The development environment is installed on the client as described in "Setting up the development environment" on page 93.
2. The POS application has been AEF-enabled as required on the 4690 system.
3. CSS is configured and running one or more virtual sessions.

In this task, you run the SampleGUI java client that connects to a SessionServer, claim an AEFSession, and display a simple GUI that allows you to interact with the session.

Running the SampleGUI client

To run the SampleGUI on a Windows client, run the command:

```
java -Djava.rmi.server.hostname=xx.xx.xx.xx -classpath
c:/si/user;c:/si/si_jars/aefsys.jar;c:/si/si_jars/siutil.jar;
com.userco.retail.client.gui.SampleGUI
```

Figure 54. Example of SampleGUI client command

where xx.xx.xx.xx is the TCP/IP address of the network interface card on the client machine that is connected to the machine running the virtual sessions, and d is the drive letter where the zip file containing this document was extracted.

If the client is setup correctly, the progress panel shown in Figure 55 is displayed while the session is retrieved and initialized.

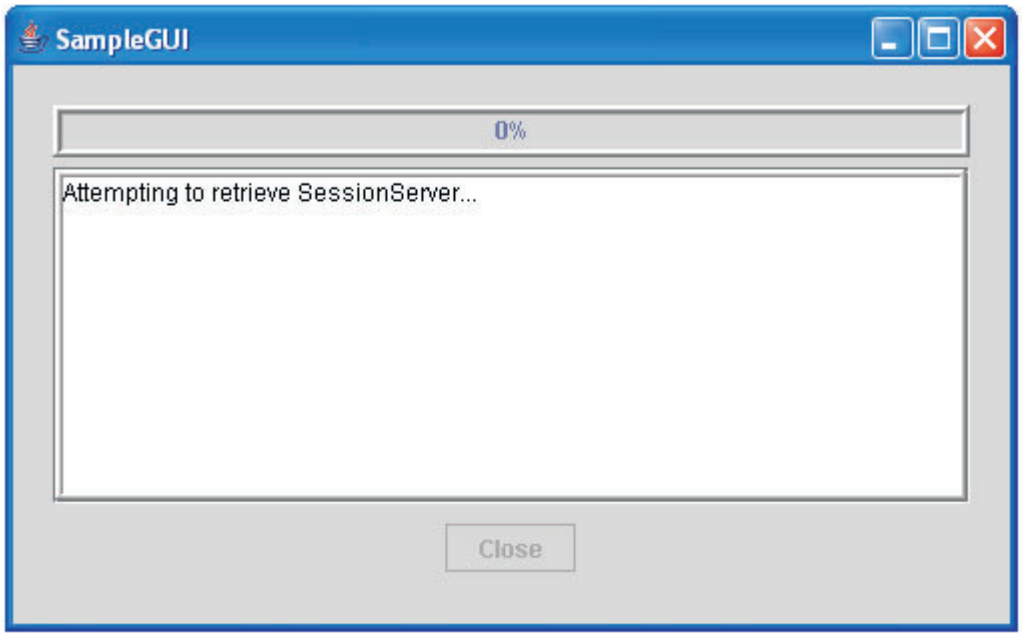

Figure 55. SampleGUI initial progress panel

Once the session has been accessed and initialized, the panel shown in Figure 56 on page 107 is displayed.

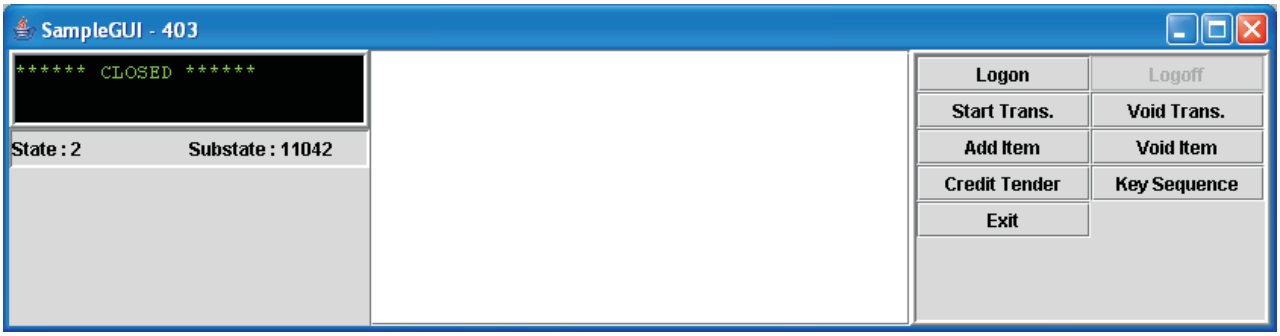

Figure 56. SampleGUI main panel

The main panel includes an operator display, a status area showing the POS application state and substate, a cash receipt area, and a button palette that allows interaction with the POSAutomationProvider API.

AEF

Infrastructure

To attach to a virtual or real session, it is necessary to find an instance of the SessionServer.

Getting a SessionServer

The simplest way to get a SessionServer is to use the AEFBase.getInstance().getSessionServer() method. This method causes a broadcast beacon to be sent out, and the first SessionServer instance that responds is used. This only locates SessionServer instances within the same TCP/IP subnet as the client. The next task describes connecting to a SessionServer outside the subnet.

This task uses the simple method of finding a SessionServer. The steps are as follows:

1. Write the AEFExample1 class, including the call to the getSessionServer method.
2. Compile the class.
3. Run the client code, assuming that CSS is running with virtual sessions on a 4690 within the same subnet as the client.

Writing the AEFExample1 class:

A complete AEFExample1 class has been provided in `c:/si/user/com/userco/retail/client/example1/AEFExample1.java`. Examine the source code and find the call to the getSessionServer method.

Compiling the code

The code has already been compiled, but if you want to modify it, you can recompile it with the following command:

```
javac -classpath .;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar
com/userco/retail/client/example1/AEFExample1
```

Executing the code

To execute the code, run the command:

```
java -Djava.rmi.server.hostname=xx.xx.xx.xx
-classpath .;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar
com.userco.retail.client.example1.AEFExample1
```

Where xx.xx.xx.xx is the TCP/IP address of the client network interface card that is on the same network as the SessionServer.

Connecting to a specific SessionServer:
This task demonstrates how to connect to a specific SessionServer, or a SessionServer that is outside of the client's TCP/IP subnet. To do this, you must:

1. Determine the URI (Uniform Resource Identifier) of the SessionServer.
2. Use the AEFBase.getRemoteServerFromURI method to fetch the SessionServer.

Determining the SessionServer URI:
The SessionServer URI form is as follows:

rmi://***xx.xx.xx.xx*:*yyyyy*/*server_name***

where

xx.xx.xx.xx is the TCP/IP address of the machine where the SessionServer is running

yyyyy is the rmi port number for the JVM running the SessionServer. This is configured in the config.properties bundle under the keys rmi.port, 4690.controller.rmi.port, and general.rmi.port. The default port for the 4690 controller is 12099. The default port for all other platforms is 11099.

server_name is the id of the SessionServer as configured in the config.properties bundle. The value is configured under the keys server.id and defaults to AEF_SESSION_SERVER***nodename*** where *nodename* is the controller node ID.

Using the getRemoteServerFromURI method:
If the SessionServer URI is "rmi://10.30.151.1:12099/AEF_SESSION_SERVERDD", then the following line may be used to attempt to connect:

```
AEFBase.getInstance().getRemoteServerFromURI
("rmi://10.30.151.1:12099/AEF_SESSION_SERVERDD");
```

Getting a specific session:
Clients, such as the remote SI GUI, might need to connect to a specific session. This is achieved through one of the SessionServer.getSession methods. If the session already exists, it is returned to the caller. If the session does not already exist, but can be created, it is returned to the client after it is created.

If the session server has been retrieved, the following lines attempt to fetch an AEFSession corresponding to session number 201.

```
AEFSession session = null;
try
{
   session = sessionServer.getSession("201");
}
catch (Throwable th)
{
   System.error.println("Unable to retrieve requested session, " + th.getMessage());
}
```

Since session initialization can take some time, there may be a delay between the time the session is created, and the time the session is ready for use. By default, the getSession method blocks until the requested session is ready for use, or until a configurable timeout occurs. This timeout value is specified in the config.properties bundle using the ready.wait.timeout key.

Note that retrieving a specific session does not stop other clients from specifically retrieving the same session. Another client could request session 201, even though the first client has already retrieved session 201.

Getting a session without a ready wait:
As discussed in the previous task, the default versions of the SessionServer.getSession and SessionServer.getAvailableSession methods block the caller until the session is ready for use, or until a timeout occurs. As an alternative, the client might want the session to be returned as soon as it is created. The client can register as a SessionStatusListener so that it can be notified when the session becomes ready.

This task demonstrates the use of a SessionParameters argument to indicate that the session should be returned immediately. It also demonstrates the use of a SessionStatusListenerProxy that provides an event indicating when the session is ready.

Examine the getAvailableSession method from the *c:*/si/user/com/userco/retail/client/gui/SampleGUI.java module:

```
1.  /**
2.  * Retrieves an available session, and does not wait for it to become ready.
3.  *
4.  * @return AEFSession
5.  *
6.  **/
7.  public static AEFSession getAvailableSession()
8.  {
9.     AEFSession retSession = null;
10.    SessionParameters parms = new SessionParameters();
11.    // A "-1" timeout value indicates that there should be no wait
12.    // for the session to become ready.  It should be returned immediately.
13.    parms.setTimeout(-1);
14.    progressFrame.apendText("Requesting available session...");
15.    try
16.    {
17.       retSession = server.getAvailableSession(parms);
18.       progressFrame.setBarValue(20);
19.       try
20.       {
21.          progressFrame.appendText("Success,  ID=" + retSession.getTerminalNumber() + ".\n");
22.       }
23.       catch (RemoteException re)
24.       {
25.          log.error("Failed to retrieve AEFSession ID.", re);
26.       }
27.
28.       // Attach a SessionStatusListener to update the progress frame.
29.       try
30.       {
31.          listenerProxy = new SessionStatusListenerProxy(retSession,
32.                                       new SessionStatusListener()
33.                {
34.                   /**
35.                    * Indicates a session status event has occurred.
36.                    *
37.                    * @param  evt contains details of SessionStatusEvent
38.                    * @throws RemoteException
39.                    *
40.                    */
41.                   public void sessionStatusChanged(SessionStatusEvent evt)
42.                   {
43.                      progressFrame.appendText(evt.getDescription() + "\n");
44.                      progressFrame.setBarValue( 20 + evt.getPercentComplete());
45.                   }
46.
47.                   /**
48.                    *  Indicates that the session is ready for commands.
49.                    *
50.                    *  @param  evt contains details of SessionStatusEvent
```

```
51.                     *  @throws RemoteException
52.                     *
53.                     */
54.                    public void sessionReady(SessionStatusEvent evt)
55.                    {
56.                       progressFrame.appendText(evt.getDescription() + "\n");
57.                       progressFrame.setBarValue( 20 + evt.getPercentComplete());
58.                       try
59.                       {
60.                          listenerProxy.removeListener();
61.                       }
62.                       catch (Throwable th2)
63.                       {
64.                          log.error("Unable to remove SessionStatusListener.", th2);
65.                       }
66.                    }
67.
68.                    /**
69.                     * Indicates that the session has ended.
70.                     *
71.                     * @param  evt contains details of SessionStatusEvent
72.                     * @throws RemoteException
73.                     *
74.                     */
75.                    public void sessionEnded(SessionStatusEvent evt)
76.                    {
77.                    }
78.                 } );
79.
80.          // If the session is already ready, we can remove the
81.          // session status listener, and update the progress frame.
82.          if (retSession.isReady())
83.          {
84.               progressFrame.appendText("Session is ready.\n");
85.               progressFrame.setBarValue( 120 );
86.               listenerProxy.removeListener();
87.           }
88.       }
89.       catch (Throwable th)
90.       {
91.          log.error("Unable to add SessionStausListener.", th);
92.       }
93.    }
94.    catch (Throwable e)
95.    {
96.       progressFrame.appendText("Failed.\n");
97.       progressFrame.appendText("   Exception : " + e.toString() + "\n");
98.       progressFrame.setButtonEnabled(true);
99.       log.error("Failed to get available session.", e);
100.   }
101
102.   return retSession;
103.}
```

On lines 10 and 13, the SessionParameters argument is created, and the session ready wait timeout value is set to -1 indicating that the session should be returned immediately. On line 17, the session is requested. On line 31, a SessionStatusListenerProxy is created. The SessionStatusListenerProxy requires an instance of a SessionStatusListener. This is supplied as an anonymous inner class starting on line 32.

Detecting that a remote device is in use:
The com.ibm.retail.AEF.util.DeviceInUseNotificator allows AEF clients to register to be notified when a remote JavaPOS device fails to open because another application is using it.

Clients of the AEF wishing to receive these notifications must implement the com.ibm.retail.AEF.io.DeviceOpenNotificationInt interface and its deviceInUse() method.

```
Public class DeviceInUseExample implements DeviceOpenNotificationInt
{
    public boolean deviceInUse(String deviceName, String applicationName)
```

```
    {
        // do something here
    }
}
```

It must create the DeviceInUseNotificator object and register to receive notifications using it's addListener() method.

```
DeviceInUseNotificator deviceInUseNotificator = new
 DeviceInUseNotificator(terminalNumber);
deviceInUseNotificator.addListener(this);
```

If a device is in use the AEF will invoke the client's deviceInUse() method, allowing it to take the appropriate action. This method of creating the DeviceInUseNotificator described above creates a com.ibm.retail.AEF.io.DeviceServer object. The DeviceInUseNotificator also provides clients with a constructor that accepts a DeviceServer in case the client wishes to create and manage the DeviceServer themselves. If the client manages the DeviceServer, it must pass the DeviceInUseNotificator a device server using this alternate constructor.

```
DeviceServer deviceServer = new DeviceServer();
DeviceInUseNotificator deviceInUseNotificator = new DeviceInUseNotificator
 (terminalNumber, deviceServer);
```

In general the DeviceInUseNotificator only notifies the client the first time a device fails, and then tries to open the device at a configured interval, configured in milliseconds via the device.open.retry.interval property in the AEF's config properties bundle, until the device opens or the client tells it to stop. To stop the retries, invoke the abortRetries() method.

```
deviceInUseNotificator.abortRetries();
```

Once the retries are aborted, the client can enable them again.

```
deviceInUseNotificator.enableRetries(true/false);
```

Sending true to this method will cause the client to receive another notification if a device is in use. If the client needs to expedite a retry, it can invoke the forceRetry() method.

```
deviceInUseNotificator.forceRetry();
```

The client can also set the retry interval to a value of their choosing.

```
deviceInUseNotificator.setRetryInterval(anInteger);
```

When the client is shutting down or has no need for the DeviceInUseNotificator any longer, it should invoke the dispose() method before clearing it's reference.

```
deviceInUseNotificator.dispose();
```

Data Provider

The Data Provider maintains a set of property values that reflect the current state of the POS application.

Retrieving a Data Provider property value

To retrieve a Data Provider value, perform the following steps:

- Determine the category and property name.
- Retrieve the POSDataProvider instance from the session.
- Use the POSDataProvider to get the property value.

Determining the category and property name

Use the tables in Appendix E, “Data Provider properties,” on page 207 to determine the property category and property name. This example shows retrieving the current transaction balance due. The category name is the symbol TransactionTotalsProperties.CATEGORY, and the property name is the symbol TransactionTotalsProperties.AMOUNT_DUE.

Retrieving the POSDataProvider instance

If the variable *session* holds a valid session, the POSDataProvider can be retrieved using the following code:

```
import com.ibm.retail.AEF.data.*;
POSDataProvider dataProvider = null;
try
{
    dataProvider = session.getPOSDataProvider();
}
catch (RemoteException re)
{
   // Some sort of network error happened.
}
```

Fetching the Property Value from the Data Provider

If the Data Provider instance has been retrieved successfully, then the property value may be retrieved with the following code:

```
String balanceDue = null;
{
try
    balanceDue = (String)(dataProvider.
    getPropertyValue(
    TransactionTotalsProperties.CATEGORY,
    TransactionTotalsProperties.AMOUNT_DUE));
}
catch (RemoteException re)
{
     // Some sort of network error happened.
}
```

SampleGUI example

For a working example of fetching a POSDataProvider property value, see the actionPerformed method in the SampleGUI module c:/si/user/com/userco/retail/client/gui/CreditTenderAction.java. In this method, the Data Provider property representing the transaction balance due is retrieved to be displayed in the dialog as shown in Figure 57:

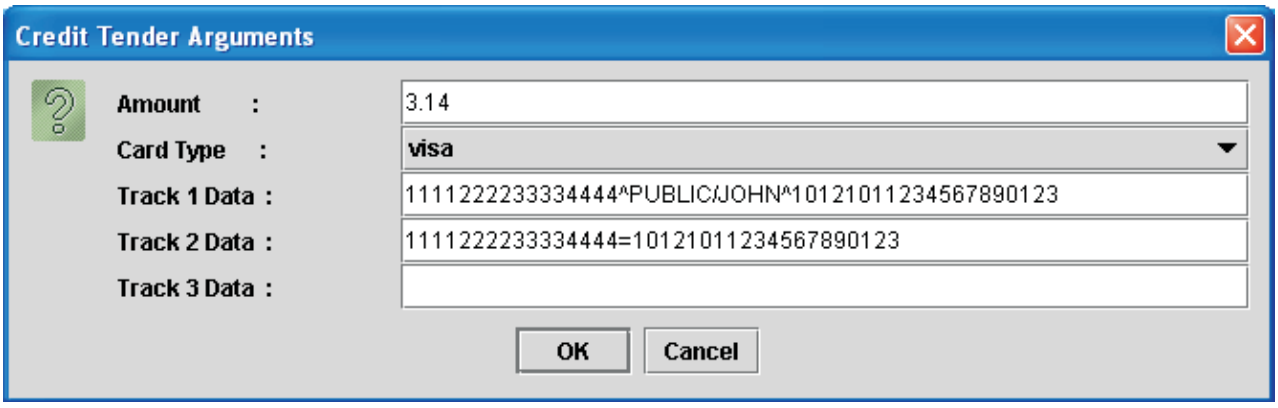

Figure 57. Displaying a Data Provider property value

Subscribing to a Data Provider event

The Data Provider publishes events triggered by the POS application. For a list of events, see “Event and listener types” on page 43. The steps for this task are as follows:

1. Create a Java class that implements the required listener interface.
2. Use an instance of the listener to create an instance of a proxy listener interface.

Creating the listener class

This example shows creating a ScaleListener. The ScaleListener can be implemented as a named, or unnamed class. This example uses a named ScaleListener called com.ibm.retail.AEF.sample.MyScaleListener. The following code for MyScaleListener must be copied to the com\ibm\retail\AEF\sample directory before compilation.

```
import com.ibm.retail.AEF.data.*;
import com.ibm.retail.AEF.event.*;
import java.rmi.*;
public class MyScaleListener implements ScaleListener
{
   /**
    *  Scale event occurred (an item was weighed).
    *
    *  @param   evt contains weight information
    *  @throws  RemoteException if remote access fails
    *
    */
   public void itemWeighed(ScaleEvent evt) throws RemoteException
   {
      // User code to process the ScaleEvent goes here.
   }
}
```

Creating an instance of the proxy listener

The proxy listener is created by passing the POSDataProvider for the session of interest, as well as an instance of the new listener class from the previous step.

```
import com.ibm.retail.AEF.client.*;
import com.ibm.retail.AEF.samples.*;
// Assume that dataProvider contains a valid reference to
// the session's data provider.
MyScaleListener myScaleListener = new MyScaleListener();
// Create the proxy (which also starts it listening).
try
{
   ScaleListenerProxy scaleProxy = new ScaleListenerProxy(
                                       dataProvider,
                                       myScaleListener);
}
catch (RemoteException re)
{
   // Network error.
}
catch (AEFException ae)
{
   // Failed to register listener with data provider.
}
// To stop the proxy from listening, use the following lines:
try
{
   scaleProxy.removeListener();
}
catch (RemoteException re)
{
   // Network error.
}
catch (AEFException ae)
{
   // Failed to remove listener from data provider.
}
```

SampleGUI example of subscribing to a POSDataProvider event

For a working example of subscribing to a POSDataProvider event, see the SampleGUI module si/user/com/userco/retail/client/gui/StatusPanel.java. This class subscribes to AEFPropertyChangeEvents to display the current application state and substate as shown in Figure 58 on page 114

page 114.

Figure 58. AEFPropertyChangeListener example

Sending additional data to an event

This task describes how to send additional information from the POS application to the AEF, and how to retrieve that information from a POSDataProvider event. For this task, it is assumed your POS application supports a time and attendance function on the operator logon and logoff operations. If the user is clocking-in during a logon, then a global variable is set to the clock-in time. In addition, another global variable is set to the clock-out time if the operator clocks-out during the logoff.

Note:

1. The sample that follows illustrates how you would write SI client code to listen for the POSDataProvider event and pull out the new time and attendance data.
2. Sample user exit code for SA and GSA is located in si\samples\sa\timeandattendance and si\samples\gsa\timeandattendance respectively.
3. For information about SA user exit programming, see the *4680-4690 Supermarket Application: Programmers Guide*. For information about the SI specific user exits, see Appendix Appendix A, "SA user exit code," on page 193.
4. For information about GSA user exit programming, see the *4680-4690 General Sales Application: Programmers Guide*. For information about the GSA specific user exits, see Appendix Appendix B, "GSA user exit code," on page 195.
5. For information about ACE extension programming, see the *SurePOS Application Client/Server Environment for 4690 OS Programming Reference*.

New SI XML message

The XML event that is sent to AEF when an operator logs on or off is modified to include time and attendance data. For example, if an operator logon includes a clock-in, the *clockIn* attribute is added to the XML as shown in the following example:

```
<signOn operatorID="887" operatorName="John Public" clockIn="07/14/2004 08:35">
  <authorization  />
</signOn>
```

In addition, if an operator logoff includes a clock-out, the *clockOut* attribute is added to the XML:

```
<signOff operatorID="887" clockOut="07/14/2004 14:58"/>
```

Retrieving the new event data

The following example shows how to retrieve the new event data out of the POSDataProvider event. No changes are necessary to the POSDataProvider, or to the relevant event classes. The additional data is automatically included in the relevant POSDataProvider events.

For this example, the clockIn and clockOut attributes are included in the POSDataProvider OperatorEvent. You can retrieve these attribute values from the OperatorEvent by using the POSAppEventElement.getProperty method.

When you code the OperatorListener.operatorEventOccurred method, you can retrieve the clockIn and clockOut attributes (if they exist) as follows:

```
public void operatorEventOccurred(OperatorEvent evt)
{
   String clockIn = null;
   String clockOut = null;
   clockIn = evt.getProperty("clockIn");
   if (clockIn != null)
   {
      // Process the clockIn here.
   }
   clockOut = evt.getProperty("clockOut");
   if (clockOut != null)
   {
      // Process the clockOut here.
   }
}
```

Sending a new XML event

This task describes how to send a completely new XML message from the POS application to the AEF. This may be necessary for AEF enabling a user extension or a feature not already AEF enabled. For this task, assume that the POS application needs to be extended to keep the AEF updated with the current tender positions of the till.

Note: The user must be familiar with the extension mechanism relevant to the POS application.

The steps required to complete this task are as follows:

- Define the XML message format.
- Modify the POS application to send the new message when appropriate.
- Implement a new Event, Listener, ListenerSupport, and ListenerProxy classes for the new event.
- Create a class that implements the SessionExtension interface. This class is responsible for registering the new ListenerSupport class with the POSDataProvider.
- Configure the Data Provider to create an instance of the new event class whenever it sees the new XML message from the POS application.
- Register to receive the event in the client. This involves using the new ListenerProxy class.
- Package and distribute the new modules.

Defining the XML message:
For this task, the XML message is in the following form:

<tillPosition cash="***nnnnn.nn***" checks=" ***nnnnn.nn***" foodstamps=" ***nnnnn.nn***"/>

Modifying the POS application to send the message:
Use the standard extension technique for the POS application to send the XML message when appropriate. The XML message must be constructed as a string.

SA and GSA Code for sending the XML message:
Sample user exit code for SA and GSA is located in si\samples\sa\tilltenderposition and si\samples\gsa\tilltenderposition, respectively.

Implementing the new event, listener, and ListenerSupport classes:
Define the following interfaces and classes:

- TillPositionEvent
- TillPositionEventImpl
- TillPositionListener
- TillPositionListenerSupport

- TillPositionListenerProxy

TillPositionEvent:
The following module:
`c:/si/user/com/userco/retail/AEF/event/TillPositionEvent.java`

can be written as follows:

```
/*
 * TillPositionEvent Interface
 *
 * 07/07/04
 *
 * Copyright:
 * Licensed Materials - Property of IBM
 * "Restricted Materials of IBM"
 * 5724-AEF
 * (C) Copyright IBM Corp. 2004, 2007.
 */

package com.userco.retail.AEF.event;
import com.ibm.retail.AEF.event.*;

/**
 *  This class is a sample user event.
 */
public interface TillPositionEvent extends POSAppEvent
{
   // Note, we want these constants to be the same as the attribute
   // names in the XML
   public static final String CASH_POSITION="cash";
   public static final String CHECK_POSITION="checks";
   public static final String FOODSTAMP_POSITION="foodstamps";
   /**
    * Get the cash position.
    *
    *  @return String
    *
    */
    public String getCashPosition();
    /**
    * Get the foodstamps position.
    *
    *  @return String
    *
    */
   public String getFoodStampPosition();
   /**
    * Get the check position.
    *
    *  @return String
    *
    */
   public String getCheckPosition();
}
```

Figure 59. TillPositionEvent

TillPositionEventImpl:
The module:

c:/si/user/com/userco/retail/AEF/event/TillPositionEventImpl.java

can be written as shown in the following code example:

```
/*
 * TillPositionEventImpl
 *
 * 07/07/04
 *
 * Copyright:
 * Licensed Materials - Property of IBM
 * "Restricted Materials of IBM"
 * 5724-AEF
 * (C) Copyright IBM Corp. 2004, 2007.
 */

package com.userco.retail.AEF.event;

import com.ibm.retail.si.util.*;
import com.ibm.retail.AEF.data.*;
import com.ibm.retail.AEF.event.*;
import org.xml.sax.*;

/**
 *  An example of a user event.
 */

public class TillPositionEventImpl extends POSAppEventImpl implements TillPositionEvent
{
   static String copyright()
      { return com.ibm.retail.si.Copyright.IBM_COPYRIGHT_SHORT; }

   /**
    *  Construct an TillPositionEventImpl
    *
    */
   public TillPositionEventImpl()
   {
      super();
   }

   /**
    *  Sets default property values.
    *
    */
   public void setDefaultProperties()
   {
      //properties.put("xxxx", new Boolean(false));
   }

   /**
    * Get the cash position.
    *
    *  @return String
    *
    */
   public String getCashPosition()
   {
      return (String)(properties.get(TillPositionEvent.CASH_POSITION));
   }
```

Figure 60. TillPositionEventImpl Part 1 of 2)

```
   /**
    * Get the foodstamps position.
    *
    *  @return String
    *
    */
   public String getFoodStampPosition()
   {
      return (String)(properties.get(TillPositionEvent.FOODSTAMP_POSITION));
   }

   /**
    * Get the check position.
    *
    *  @return String
    *
    */
   public String getCheckPosition()
   {
      return (String)(properties.get(TillPositionEvent.CHECK_POSITION));
   }

   /**
    *  Gets the AEF property change category defined for this event.
    *  @return String   category id
    *
    */
   public String getPropertyChangeCategory()
   {
      return WorkstationStatusProperties.CATEGORY;
   }
}
```

Figure 61. TillPositionEventImpl Part 2 of 2)

TillPositionListener:
The module

c:/si/user/com/userco/retail/AEF/event/TillPositionListener.java

can be written as follows:

```
/*
* TillPositionListener Interface
*
* 07/07/04
*
* Copyright:
* Licensed Materials - Property of IBM
* "Restricted Materials of IBM"
* 5724-AEF
* (C) Copyright IBM Corp. 2004, 2007.
*
*/
package com.userco.retail.AEF.event;
import com.ibm.retail.AEF.event.*;
import java.rmi.*;
/**
* Listener interface for receiving <code>TillPositionEvents</code> from the
* terminal session.
*
*/
public interface TillPositionListener extends POSAppEventListener
{
/**
 * The till contents have changed.
```

```
 *
 * @param evt
 *
 */
 public void tillPositionChanged(TillPositionEvent evt) throws RemoteException;
}
```

TillPositionListenerSupport:
The following module helps the POSDataProvider determine the listener type and listener method to call for a specific event type.

c:/si/user/com/userco/retail/AEF/event/TillPositionListenerSupport.java

The module can be written as follows:

```
/*
 * TillPositionListenerSupport Interface
 *
 * 07/07/2004
 *
 * Copyright:
 * Licensed Materials - Property of IBM
 * "Restricted Materials of IBM"
 * 5724-AEF
 * xxxx-xxx
 * (C) Copyright IBM Corp. 2004, 2007.
 *
 */
package com.userco.retail.AEF.event;
import java.util.*;
import java.rmi.*;
import com.ibm.retail.AEF.util.*;
import com.ibm.retail.AEF.event.*;
import com.ibm.retail.si.util.*;
import com.ibm.retail.si.util.AEFConst;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
/**
*  ListenerSupport objects are used internally by the POSDataProvider
*  to handle notification of events received.
*/
public class TillPositionListenerSupport extends POSAppEventListenerSupport
{   private static Log log = LogFactory.getLog(TillPositionListenerSupport.class);
   /**
    *  Add a listener to be notified of events handled by this support class
    *
    *  @param  listener
    *  @throws AEFException if listener is not of proper type
    *
    */
   public void addPOSAppEventListener(POSAppEventListener listener) throws AEFException
   {
      if (listener instanceof TillPositionListener)
      {
         super.addPOSAppEventListener(listener);
         if (log.isTraceEnabled())
         {
            log.trace("Added listener");
         }
      }
      else
      {
         throw (new AEFException(AEFConst.INVALID_LISTENER_TYPE, listener.getClass().getName()));
      }
   }
   /**
    *  Notify an event listener
    *
    *  @param listener
    *  @param event
```

```
    *  @throws AEFException if notification fails
    *
    */
   public void notifyListener(POSAppEventListener listener, POSAppEvent evt) throws AEFException
   {
      if (evt instanceof TillPositionEvent)
      {
         TillPositionListener l = (TillPositionListener)listener;
         try
         {
            if (log.isTraceEnabled())
            {
               log.trace("Notify listener: " + evt.toString() );
            }
            l.tillPositionChanged((TillPositionEvent)evt);
         }
         catch (Exception e)
         {
            handleListenerException(listener, e);
         }
      }
      else
      {
         // improper event type - throw an exception
         throw (new AEFException(AEFConst.EVENT_LISTENER_MISMATCH, evt.getClass().getName()));
      }
   }
   /**
    *  Returns class name of listeners that are supported
    *
    *  @return listener class name
    *
    */
   public String getListenerClassName()
   {
      return (TillPositionListener.class.getName());
   }
   /**
    *  Returns class name of POSAppEvents that are supported
    *
    *  @return event class name
    *
    */
   public String getEventClassName()
   {
      return (TillPositionEventImpl.class.getName());
   }
}
```

The four modules above can be compiled with the following command:

```
javac -classpath

c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar -d c:/si/userjar
com/userco/retail/AEF/event/*.java
```

TillPositionListenerProxy:
The module *c:*/si/user/com/userco/retail/AEF/client/TillPositionListenerProxy.java is an RMI object that helps localize event dispatching in a remote client. This module requires a special compile step since it is an RMI object. The module can be written using the following code:

```
/*
 * TillPositionListenerProxy
 *
 * 07/074/04
 *
 * Copyright:
 * Licensed Materials - Property of IBM
```

```
 * "Restricted Materials of IBM"
 * 5724-AEF
 * (C) Copyright IBM Corp. 2004, 2007.
 */
package com.userco.retail.AEF.client;

import com.ibm.retail.AEF.event.*;
import com.ibm.retail.AEF.data.*;
import com.ibm.retail.AEF.client.*;
import com.ibm.retail.AEF.util.*;
import com.ibm.retail.si.util.*;
import com.ibm.retail.si.Copyright;
import com.userco.retail.AEF.event.*;
import java.rmi.*;
import java.rmi.server.*;
import java.lang.reflect.*;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

/**
* The TillPositionListenerProxy provides a proxy object for remote listeners
* to monitor the TillPosition events of its associated terminal session.
* <p>
* Remote proxy objects extend the appropriate RMI server classes which handle the
* RMI duties for the client (so the client does not need to deal with rmi server issues).
* The proxy objects also perform the role of event dispatching.
* <p>
* By default, the listener proxy object will perform event queuing on an AEF event thread.
* This relieves the client of any "thread swapping" responsibilities and insures that the
* AEF event dispatching performance is not affected by client processing.  To override this
* default behavior, use the <i>setDispatchQueue()</i> method.  The proxy utilizes the
* EventDispatcher for queueing event listener notification.
* <p>
* To use a proxy object, the client must implement the listener interface
* and get the POSDataProvider object from the AEFSession to monitor. The proxy
* object is then constructed passing a reference to the client and the data provider.
* The proxy object performs the listener registration and forwards
* all events from the data provider to the client.
*
*/

public class TillPositionListenerProxy extends AEFEventListenerProxy implements
             AEFListenerProxyInterface,TillPositionListener
{
   static String copyright()
      { return com.ibm.retail.si.Copyright.IBM_COPYRIGHT_SHORT; }

   private static Log log = LogFactory.getLog(TillPositionListenerProxy.class);
   protected static Method listenerMethod = getListenerMethod(
           TillPositionEvent.class,TillPositionListener.class,
           "tillPositionChanged");

   /**
    * Construct an TillPositionListenerProxy.
    * This method registers the proxy as a listener with the POSDataProvider
    *and begins event notification to the proxy client listener.
    *
    * @param dataProvider instance of POSDataProvider which is the source of
    *                     events
    * @param listener client listener which receives the event notification via
    *                 this proxy
    *
    * @exception java.rmi.RemoteException
```

```
     * @exception AEFException
     *    Among the possible AEFException error codes are:
     *       <br>AEFConst.NO_LISTENER_SUPPORT
     *
     */
    public TillPositionListenerProxy(POSDataProvider dataProvider,
            TillPositionListener listener) throws RemoteException, AEFException
    {
       this.dataProvider = dataProvider;
       this.listener     = listener;
       dataProvider.addPOSAppEventListener(TillPositionListener.class.
                                        getName(),this);
    }

    /**
     * Remove the proxy listener from the POSDataProvider. Ends notification of
     * events.
     *
     * @exception java.rmi.RemoteException
     * @exception AEFException
     *    Among the possible AEFException error codes are:
     *       <br>AEFConst.NO_LISTENER_SUPPORT
     *
     */
    public void removeListener() throws RemoteException, AEFException
    {
       dataProvider.removePOSAppEventListener(TillPositionListener.class.
                                           getName(),this);
    }

    /**
     * Till position changed on the POS terminal session.
     * Dispatches event notification to the proxy client listener.
     *
     * @param   evt
     * @throws  RemoteException if remote access fails
     *
     */
    public void tillPositionChanged(TillPositionEvent evt) throws RemoteException
    {
       dispatchEvent(evt,listener,listenerMethod);
    }

    // instance data
    protected POSDataProvider        dataProvider;
    protected TillPositionListener       listener;
}
```

The TillPositionListenerProxy must be compiled using the following two commands:

```
javac -classpath .;c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar -d c:/si/userjar
com/userco/retail/AEF/client/*.java

rmic -classpath .;c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar -d c:/si/userjar
com.userco.retail.AEF.client.TillPositionListenerProxy
```

Create a SessionExtension to Register the ListenerSupport Object:
An instance of the new TillPositionListenerSupport class must be registered with the POSDataProvider. Make use of the SessionExtension to accomplish this. Classes that implement the SessionExtension

interface may be configured to be created when the session is created. An instance of the class is created, and a reference is passed to the session. Use this feature to register an instance of the new TillPositionListenerSupport class.

Create the class file using the following code:

c:/si/user/com/userco/retail/AEF/session/UserListenerSupportRegistrar.java

Include the following contents:

```
/*
 * UserListenerSupportRegistrar
 *
 * 07/07/04
 *
 * Copyright:
 * Licensed Materials - Property of IBM
 * "Restricted Materials of IBM"
 * 5724-AEF
 * xxxx-xxx
 * (C) Copyright IBM Corp. 2004, 2007.
 *
 */
package com.userco.retail.AEF.session;

import com.userco.retail.AEF.event.*;
import com.ibm.retail.AEF.data.*;
import com.ibm.retail.AEF.session.*;

import java.rmi.*;

/**
 * This class is configured to be created when the session is created.
 * We use it to register our TillPositionListenerSupport instance with the
 * POSDataProvider.
 */

public class UserListenerSupportRegistrar implements SessionExtension
{
   /**
    * Constructor
    *
    *
    */
   public UserListenerSupportRegistrar()
   {
      tillPositionListenerSupport = new TillPositionListenerSupport();
   }

   /**
    * This method will be called after the session becomes ready.
    *
    * @param session The AEFSession associated with this extension object.
    *
    */
   public void setSession(AEFSession session)
   {
      POSDataProvider dataProvider = null;
      try
      {
         dataProvider=session.getPOSDataProvider();
         dataProvider.addPOSAppEventListenerSupport(tillPositionListenerSupport);
      }
      catch (RemoteException re)
      {
         // Impossible to get here, because the call is local.
      }
```

```
   }

   // Instance variables
   protected TillPositionListenerSupport tillPositionListenerSupport;

}
```

This module can be compiled with the following command:

```
javac -classpath .;c:/si/si_jars/aefsa.jar;
c:/si/si_jars/aefsys.jar;c:/si/si_jars/siutil.jar -d c:/si/userjar
com/userco/retail/AEF/session/*.java
```

You must configure the AEF to create an instance of the new UserListenerSupportRegistrar class by creating or editing the file c:/si/userjar/usersession.properties and adding the following line:

```
session.extensions=com.userco.retail.AEF.session.UserListenerSupportRegistrar
```

Configuring Data Provider to create the new event:
To configure the Data Provider to create and fire an instance of the new TillPositionEventImpl class, you must add the XML element tag tillPosition as a key in the classes.properties bundle. To do this, create or edit the following file:

```
c:/si/userjar/userclass.properties
```

and add the line:

```
tillPosition=com.userco.retail.AEF.event.TillPositionEventImpl
```

Registering the client for the new event:

The client should use the new TillPositionListenerProxy to receive and process the new TillPositionEvents. The following code fragment shows how a client would create a TillPositionListenerProxy:

```
// Attach a TillPositionListener to process the TillPositionEvents.
// Assume that "session" holds a valid AEFSession.
      TillPositionListenerProxy listenerProxy = null;
      try
      {
         listenerProxy = new
                     TillPositionListenerProxy(session,
                                       new TillPositionListener()
                {
                   /**
                    * Indicates till position has changed.
                    *
                    * @param  evt contains details of TillPositionEvent
                    *
                    */
                    public void tillPositionChanged(TillPositionEvent evt)
                    {
                       // Process the new till position here!
                       System.out.println("New cash position = " +
                                          evt.getCashPosition();
                    }
                }
                                          );
      }
      catch (Throwable th)
      {
         log.error("Unable to add TillPositionListener.", th);
      }
```

To create a new user.jar file, issue the following command:

```
jar cfv user.jar *
```

Packaging the new modules:
The user class and properties files need to be bundled into the appropriate user*.jar file, copied into the 4690 c:\adx_ipgm directory, and distributed. This might require that CSS and the registers be stopped. The jar file also needs to be included on the classpath in the client environment.

Automation Provider

Tasks are described in the following sections to implement the Automation Provider.

Using an AEFSession

This task illustrates the use of an AEFSession. After acquiring the session, the example shows logging onto the session, starting a transaction, adding an item, suspending the transaction, and logging off the session.

The steps involved in this task are as follows:

- Write the AEFExample3 class, that gets an AEFSession, and use the POSAutomation API to initialize the session, logon to the session, start a transaction, add an item to the transaction, suspend the transaction, then logoff the session.
- Compile the class.
- Run the client code, assuming that CSS is running with virtual sessions on a 4690 within the same subnet as the client.

Note: This example uses default operator id and passwords to logon. Therefore these must be configured as described in "Default operator ID/passwords" on page 181.

Writing the AEFExample3 class

A complete AEFExample3 class has been provided in c:/si/user/com/userco/retail/client/example3/AEFExample3.java. Examine the source code to see how the AEFSession is used.

Compiling the code

The code has already been compiled, but if you want to modify it, you can recompile it with the following command:

```
javac -classpath c:/si/si_jars/aefsys.jar;c:/si/si_jars/siutil.jar
com/userco/retail/client/example3/AEFExample3.java
```

Executing the code

To execute the code, run the following command:

```
java -Djava.rmi.server.hostname=xx.xx.xx.xx -classpath .;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar
com.userco.retail.client.example1.AEFExample3
```

Where *xx.xx.xx.xx* is the TCP/IP address of the client network interface card that is on the same network as the SessionServer.

When the program is run, its output is displayed on the console window.

Using an AEFSession SampleGUI

The main panel of the SampleGUI application contains a button palette. Each button is associated with an action, and many of these actions use POSAutomationProvider API methods to interact with the AEFSession. See Figure 62 on page 126.

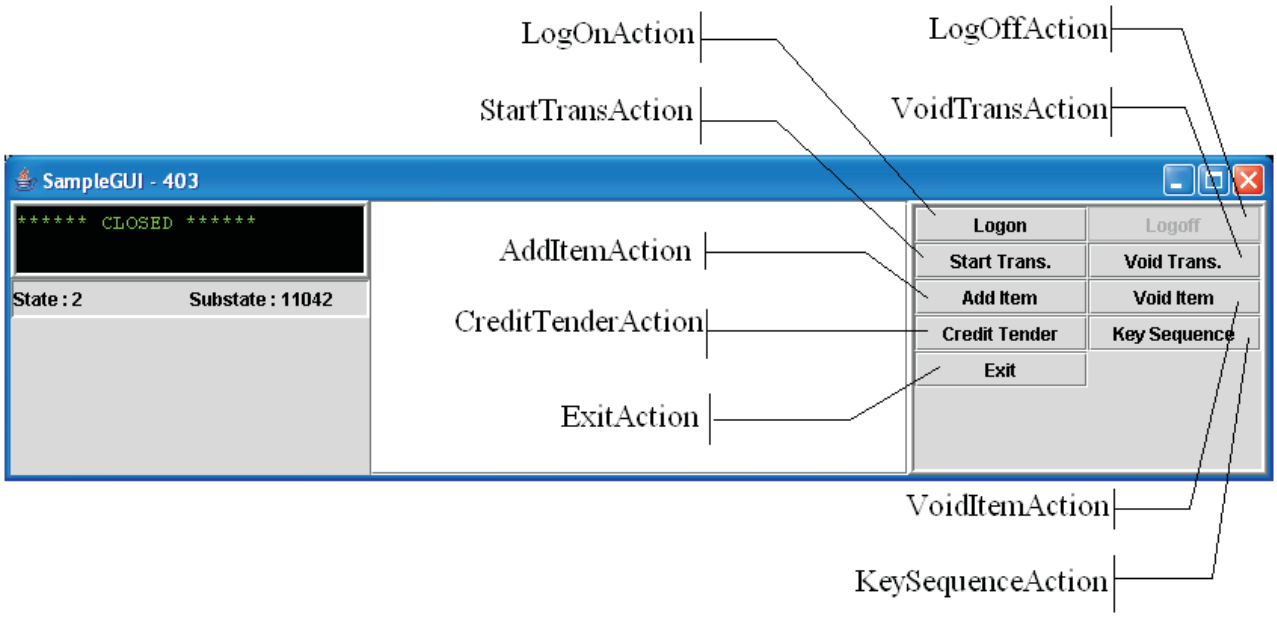

Figure 62. SampleGUI action classes

Each of the action classes in the figure above call a method in c:/si/user/com/userco/retail/client/gui/Controller.java. This class demonstrates using POSAutomationProvider methods to interact with an AEFSession. For example, the LogonAction calls the Controller.logon method as shown on line 111 of the following code:

```
1. /*
2.  * Controller
3.  *
4.  * 07/07/04
5.  *
6.  * Copyright:
7.  * Licensed Materials - Property of IBM
8.  * "Restricted Materials of IBM"
9.  * 5724-AEF
10.  * (C) Copyright IBM Corp. 2004, 2007.
11.  *
12.  */
13. package com.userco.retail.client.gui;
14.
15. import com.ibm.retail.AEF.session.*;
16. import com.ibm.retail.AEF.automation.*;
17. import com.ibm.retail.AEF.data.*;
18. import com.ibm.retail.AEF.workstation.*;
19. import java.util.logging.*;
20. import java.util.*;
21. import java.lang.ref.*;
22.
23. import com.ibm.retail.si.Copyright;
24.
25. import org.apache.commons.logging.Log;
26. import org.apache.commons.logging.LogFactory;
27.
28. /**
29.  * This class is used as the controller interface to the AEF Automation API.
30.  * All SampleGUI calls to the automation API go through the controller.
31.  */
32. public class Controller
33. {
```

```
34.
35.    static String copyright()
36.    { return com.ibm.retail.si.Copyright.IBM_COPYRIGHT_SHORT; }
37.
38.    private static Log log = LogFactory.getLog(Controller.class);
39.
40.
41.   /**
42.    * Default constructor.
43.    *
44.    * @param boolean True if the requested session was a specific
45.    *                      terminal number.  False if the session was
46.    *                      any available session.
47.    */
48.    public Controller(boolean specificSession)
49.    {
50.       specific = specificSession;
51.    }
52.
53.   /**
54.    * Sets the session.
55.    *
56.    * @param session
57.    */
58.    public void setSession(AEFSession session)
59.    {
60.       this.session = session;
61.       try
62.       {
63.          automation = session.getPOSAutomationProvider();
64.       }
65.       catch (Throwable th)
66.       {
67.          log.error("Unable to retrieve POSAutomationProvider.",th);

68.       }
69.    }
70.
71.   /**
72.    * Gets the session.
73.    *
74.    * @return AEFSession
75.    */
76.    public AEFSession getSession()
77.    {
78.       return session;
79.    }
80.
81.   /**
82.    * Gets the transaction if one is in progress, otherwise returns null.
83.    *
84.    * @return Transaction
85.    */
86.    public Transaction getTransaction()
87.    {
88.       Transaction retVal = null;
89.       if (automation == null)
90.       {
91.          log.error("Cannot get transaction because POSAutomationProvider API is unavailable.");
92.       }
93.       else
94.       {
95.          try
```

```
96.          {
97.             retVal = automation.getTransaction();
98.          }
99.          catch (Throwable th)
100.         {
101.            log.error("Unable to retrieve transaction from POSAutomationProvider.", th);
102.         }
103.      }
104.      return retVal;
105.   }


106.   /**
107.    * Logs onto the session with the OperatorIdentifer arguments.
108.    *
109.    * @return Operator
110.    */
111.    public Operator logon(OperatorIdentifier opIdent)
112.    {
113.       Operator retVal = null;
114.
115.       if (automation == null)
116.       {
117.          log.error("Cannot logon because POSAutomationProvider API not available.");
118.       }
119.       else
120.       {
121.       try
122.          {
123.             retVal = automation.logon(opIdent);
124.          }
125.          catch (Throwable th)
126.          {
127.             // Perform error handling here.
128.             log.error("Unable to logon.", th);
129.          }
130.       }
131.       return retVal;
132.    }
133.
134.   /**
135.    * Logs off the session.
136.    *
137.    */
138.    public void logoff()
139.    {
140.       if (automation == null)
141.       {
142.          log.error("Cannot logoff because POSAutomationProvider API not available.");
143.       }
144.       else
145.       {
146.          try
147.          {
148.             automation.logoff();
149.          }
150.          catch (Throwable th)
151.          {
152.             // Perform error handling here.
153.             log.error("Unable to logoff.", th);
154.          }
155.       }
156.    }
```

```
157.
158.   /**
159.    * Starts a regular sales transaction.
160.    *
161.    * @return SalesTransaction
162.    */
163.    public SalesTransaction startTransaction(TransactionIdentifier transIdent)
164.    {
165.       SalesTransaction retVal = null;
166.       if (automation == null)
167.       {
168.          log.error("Cannot start transaction because POSAutomationProvider API not available.");
169.       }
170.       else
171.       {
172.          try
173.          {
174.             retVal = (SalesTransaction)(automation.startTransaction(transIdent));
175.          }
176.          catch (Throwable th)
177.          {
178.             // Perform error handling here.
179.             log.error("Unable to start transaction.", th);
180.          }
181.       }
182.       return retVal;
183.     }
184.
185.   /**
186.    * Voids the transaction in progress.
187.    *
188.    */
189.    public void voidTransaction()
190.    {
191.       if (automation == null)
192.       {
193.          log.error("Cannot void transaction because POSAutomationProvider API not available.");
194.       }
195.       else
196.       {
197.          try
198.          {
199.              automation.voidCurrentTransaction();
200.          }
201.          catch (Throwable th)
202.          {
203.             // Perform error handling here.
204.             log.error("Unable to void transaction.", th);
205.          }
206.       }
207.    }
208.
209.   /**
210.    * Releases the session.
211.    *
212.    */
213.    public void releaseSession()
214.    {
215.       if (!specific)
216.       {
217.          // We only need to release the session if it was gotten
218.          // via a call to getAvailableSession.
219.          try
```

```
220.          {
221.             session.release();
222.          }
223.          catch (Throwable th)
224.          {
225.             // Perform error handling here.
226.             log.error("Unable to release the session.", th);
227.          }
228.       }
229.    }
230.
231.   /**
232.    * Adds one or more items to the sales transaction.
233.    *
234.    * @param itemIdent The ItemIdentifier containing arguments such
235.    * as the itemcode, quantity, etc.
236.    * @return ArrayList of LineItems.
237.    */
238.    public ArrayList addItem(ItemIdentifier itemIdent)
239.    {
240.       ArrayList retVal = null;
241.       if (automation == null)
242.       {
243.          log.error("Cannot add items because POSAutomationProvider API not available.");
244.       }
245.       else
246.       {
247.          SalesTransaction trans = (SalesTransaction)(getTransaction());
248.          if (trans == null)
249.          {
250.             log.error("Cannot add items because a transaction is not in progress.");
251.          }
252.          else
253.          {
254.             try
255.             {
256.                retVal = trans.addItem(itemIdent);
257.             }
258.             catch (Throwable th)
259.             {
260.                // Perform error handling here.
261.                log.error("Unable to add item.", th);
262.             }
263.          }
264.       }
265.       return retVal;
266.    }
267.
268.   /**
269.    * Voids an item from the sales transaction.
270.    *
271.    * @param itemIdent The ItemIdentifier containing arguments such
272.    * as the itemcode, quantity, etc.
273.    */
274.    public void voidItem(ItemIdentifier itemIdent)
275.    {
276.       if (automation == null)
277.       {
278.          log.error("Cannot add items because POSAutomationProvider API not available.");
279.       }
280.       else
281.       {
282.          SalesTransaction trans = (SalesTransaction)(getTransaction());
```

```
283.          if (trans == null)
284.          {
285.             log.error("Cannot void item because a transaction is not in progress.");
286.          }
287.          else
288.          {
289.             try
290.             {
291.                trans.voidItem(itemIdent);
292.             }
293.             catch (Throwable th)
294.             {
295.                // Perform error handling here.
296.                log.error("Unable to void item.", th);
297.             }
298.          }
299.       }
300.    }
301.
302.    /**
303.     * Adds a tender to the sales transaction.
304.     *
305.     * @param creditIdent The MSRCreditIdentifier
306.     * containing arguments such
307.     * as the tender amount, card type and track data.
308.     * @return ArrayList of LineItems.
309.     */
310.    public ArrayList addTender(MSRCreditIdentifier creditIdent)
311.    {
312.       ArrayList retVal = null;
313.       if (automation == null)
314.       {
315.          log.error("Cannot add tender because POSAutomationProvider API not available.");
316.       }
317.       else
318.       {
319.          SalesTransaction trans = (SalesTransaction)(getTransaction());
320.          if (trans == null)
321.          {
322.             log.error("Cannot add items because a transaction is not in progress.");
323.          }
324.          else
325.          {
326.             try
327.             {
328.                retVal = trans.addTender(creditIdent);
329.             }
330.             catch (Throwable th)
331.             {
332.                // Perform error handling here.
333.                log.error("Unable to add tender.", th);
334.             }
335.          }
336.       }
337.       return retVal;
338.    }
339.
340.    /**
341.     * Sends a key sequence to the POS application.
342.     *
343.     * @param sequence    The key sequence to be sent.  Function codes must
344.     * be inside < > characters.
345.     */
```

```
346.    public void keySequence(String sequence)
347.    {
348.       Workstation workstation = null;
349.       try
350.       {
351.          workstation = session.getWorkstation();
352.       }
353.       catch (Throwable th)
354.       {
355.          log.error("Unable to retrieve Workstation to send the key sequence.", th);
356.          return;
357.       }
358.       try
359.       {
360.          workstation.sendKeySequence(sequence);
361.       }
362.       catch (Throwable th2)
363.       {
364.          log.error("Unable to send key sequence to POS application.", th2);
365.       }
366.    }
367.
368.    // Member variables
369.    protected AEFSession            session;
370.    protected POSAutomationProvider automation;
371.    protected boolean               specific;
372. }
```

Extending existing Automation Provider APIs

Because of user enhancements or modifications to the POS application, it might be necessary to extend existing POSAutomationProvider APIs to handle additional arguments. For this task, the start transaction key sequence of the POS application has been modified to include a customer Zipcode.

The steps necessary for this task are as follows:

1. Add the additional zipcode argument to the TransactionIdentifier used to start the transaction.
2. Replace the implementation of SAStartTransactionActionImpl.java to use the additional data.
3. Modify the key sequence to override the start transaction key sequence definition.
4. Bundle the new files in a user.jar file, install and distribute on the 4690.

Including additional API arguments

Assume that the client starts a transaction with the following code:

```
TransactionIdentifer transIdentifier = new
                     TransactionIdentifierImpl();
transIdent.setTransactionType(AEFConst.REGULAR_SALE);
SalesTransaction tran = null;
try
{
   tran = automation.startTransaction(transIdentifier);
}
catch (RemoteException re)
{
   // Handle the network problem here.
}
catch (AEFException ae)
{
   // Handle the AEF problem here.
}
```

You can take advantage of the fact that the TransactionIdentifier is a HashMap to add the additional argument as follows:

```
String zipcode = "27607";
TransactionIdentifer transIdentifier = new
                     TransactionIdentifierImpl();
transIdent.setTransactionType(AEFConst.REGULAR_SALE);
transIdent.put("zipcode", zipcode);
SalesTransaction tran = null;
try
{
   tran = automation.startTransaction(transIdentifier);
}
catch (RemoteException re)
{
   // Handle the network problem here.
}
catch (AEFException ae)
{
   // Handle the AEF problem here.
}
```

Implementing the action

Next, we look at the JavaDoc for the startTransaction method to see that the Action ID for the method is StartTransaction. For this example, the POS application is SA and the implementing class is SAStartTransactionActionImpl. Override this class to do the following:

1. In the constructor, retrieve and save the **Zipcode** from the TransactionIdentifier.
2. In the sendStartTransactionSequence method, add the **Zipcode** to the key sequence arguments.

To achieve these two steps, you must create a new class called:

```
c:/si/user/com/userco/retail/AEF/action/UserStartTransactionActionImpl.java
```

This class extends SAStartTransactionActionImpl.java as shown in the following code example:

```
1. /*
2.  * UserStartTransactionActionImpl
3.  *
4.  * 07/07/2004
5.  *
6.  * Copyright:
7.  * Licensed Materials - Property of IBM
8.  * "Restricted Materials of IBM"
9.  * 5724-AEF
10.  * (C) Copyright IBM Corp. 2004, 2007.
11.  *
12.  */
13. package com.userco.AEF.action;
14.
15. import com.ibm.retail.AEF.automation.*;
16. import com.ibm.retail.AEF.util.*;
17. import com.ibm.retail.si.util.*;
18. import com.ibm.retail.si.Copyright;
19. import com.ibm.retail.AEF.factory.*;
20. import com.ibm.retail.AEF.thread.*;
21. import com.ibm.retail.AEF.data.*;
22. import com.ibm.retail.AEF.session.*;
23. import com.ibm.retail.AEF.action.*;
24.
25. import java.util.*;
26. import java.rmi.*;
27.
28. import org.apache.commons.logging.Log;
29. import org.apache.commons.logging.LogFactory;
30.
31. /**
32.  * UserStartTransactionActionImpl overrides SAStartTransactionActionImpl
33.  * to handle an additional zipcode argument on the start transaction
```

```
34.  * key sequence.
35.  *
36.  **/
37. public class UserStartTransactionActionImpl extends SAStartTransactionActionImpl
38. {
39.
40.    static String copyright()
41.       { return com.ibm.retail.si.Copyright.IBM_COPYRIGHT_SHORT; }
42.    private static AEFPerfTrace perfTrace  = AEFPerfTrace.getInstance();
43.
44.   /**
45.    * Constructor
46.    *
47.    * @param request The ActionRequest.
48.    * @exception AEFException
49.    * AEFException error codes:
50.    * <br>AEFConst.INVALID_ARGUMENT, AEFConst.INVALID_TRANSACTION_TYPE
51.    *
52.    */
53.    public UserStartTransactionActionImpl(ActionRequest request) throws AEFException
54.    {
55.       super(request);
56.
57.       TransactionIdentifier transactionID = (TransactionIdentifier)
           (request.getArguments());
58.       zipcode = (String)(transactionID.get("zipcode"));
59.       if (zipcode == null || zipcode.length() == 0)
60.       {
61.          throw new AEFException(AEFConst.INVALID_ARGUMENT,
62.                                 -9000,  // -9000 to -9999 reserved for user error codes
63.                                 "UserStartTransactionActionImpl.constructor(): zipcode is a
                                      required argument.");
64.       }
65.    }
66.
67.   /**
68.    * Sends the key sequence which will cause the transaction to be started.
69.    *
70.    * @return int An integer value representing the result of the key sequence.
71.    * @exception AEFException
72.    */
73.    public int sendStartTransactionSequence() throws AEFException
74.    {
75.
76.       int retVal = -1;
77.       args.clear();
78.       args.put("SEQUENCE_ID", "startTransaction");
79.       args.put("%0", zipcode);
80.       keySequenceAction = (AEFAction)(actionFactory.makeAction(new ActionRequest("SimpleKeySequence
            Action", args)));
81.       Condition[] goodConditions =
82.       {
83.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
84.                                      POSDeviceProperties.POS_SUB_STATE,
85.                                      Substate.getSubstate("SELECT_PROCEDURE")),
86.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
87.                                      POSDeviceProperties.POS_SUB_STATE,
88.                                      Substate.getSubstate("TRAINING_MODE")),
89.       };
90.       lock = new ConditionLock();
91.       detector=((DetectorAccess)(SessionContext.getSession())).
92.                                getTransactionDetector();
93.       // Save any current transaction instance number so we are sure to only get a transaction
94.       // created after this key sequence is sent.
95.       instanceNumber = detector.getInstanceNumber();
96.
97.       retVal = lock.performActionAndWait("wait-for-expecting-item-entry-substate",
98.                                          keySequenceAction,
99.                                          goodConditions,
100.                                      BadConditionsImpl.
101.                                         getInstance().getBadConditions(),
102.                                      getTimeout());
```

```
103.      if (retVal < 0)
104.      {
105.         AEFErrorHandler errorHandler = new AEFErrorHandler("Send Start Transaction Sequence Error");
106.         retVal = errorHandler.handleError(lock,
107.                                           keySequenceAction,
108.                                           goodConditions,
109.                                        BadConditionsImpl.
110.                                          getInstance().getBadConditions(),
111.                                        getTimeout(),
112.                                        retVal);
113.      }
114.      return retVal;
115.   }
116.
117.   /* Instance Variables */
118.   protected String                   zipcode;
119.   private static Log log =
120.         LogFactory.getLog(UserStartTransactionActionImpl.class);
121. }
```

This class overrides the constructor to pull out the Zipcode from the arguments as shown on line 58. The sendStartTransactionSequence method is exactly the same as in the parent SAStartTransactionActionImpl class, with the exception of line 79, that adds the Zipcode value to the key sequence to be sent to the application, and the removal of some tracing statements.

After making sure the directory d:\si\userjar exists, this class may be compiled with the following command:

```
javac -classpath
c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;c:/si/si_jars/siutil.jar -d
c:/si/userjar
com/userco/retail/AEF/action/UserStartTransactionActionImpl.java
```

Modifying the key sequence:
As shown on line 78 in the source code, the key sequence is defined by the key startTransaction. For this task, you must redefine this to be 0<SLASH>{0}<ENTER>. The {0} is a substitution variable that is filled in with the zipcode value as shown on line 79 in the source code. Note that if you needed a second substitution variable, specify it as {1}, and the code to fill in the value would use the key %1.

To override the key sequence, create or edit the file c:/si/userjar/appseq.properties. Add the following line:

```
startTransaction=0<SLASH>{0}<ENTER>
```

Bundling the files:
The new UserStartTransactionActionImpl.class and userseq.properties files must be included in the appropriate user*.jar file, placed on the 4690, and distributed. This may require CSS and/or the terminals to be stopped.

To create a new user.jar file, issue the following command:

```
jar cfv user.jar *
```

To update an existing user.jar file to add the two new files, issue the command:

```
jar uvf user.jar com/userco/retail/AEF/action/UserStartTransactionActionImpl.class userseq.properties
```

Extending the Automation Provider API with new function:
The Automation Provider includes an extension mechanism that applies when none of the existing APIs can serve the purpose required by the extension. This mechanism makes use of the *action* model already discussed. In general, a new action class is written that performs the required function and returns a result. The steps required are as follows:

- Create a new action class to perform the new function.
- Configure a class key in userclass.properties.

- Package the new class file and properties files in a user*.jar file as appropriate.
- Perform the new action by calling the POSAutomationProvider.performAction method to perform the new action and return a result.

Creating a new action class:
The new action class must implement the AEFAction interface. For example, if you want a new action that performs a gift card activation function, create the following Java source file:

```
c:/si/user/com/userco/retail/AEF/action/GiftCardActivationImpl.java
```

This class must implement the AEFAction interface, and define a performAction method. If a result is to be returned to the client, it is returned from the performAction method. Note that the object returned must be either *Serializable* or an *RMI* object.

After making sure the directory d:\si\userjar exists, this class may be compiled with the following command:

```
javac -classpath c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar -d c:/si/userjar
com/userco/retail/AEF/action/GiftCardActivationImpl.java
```

Creating the class key:
A new class key for the new action class must be added to userclass.properties. Edit (or create) the file c:/si/userjar/userclass.properties, and add the following line:

```
GiftCardActivationAction=com.userco.retail.AEF.action.GiftCardActivationActionImpl
```

Packaging the files:
The new GiftCardActivationActionImpl.class and userclass.properties files must be included in the appropriate user*.jar file, placed on the 4690, and distributed. This may require CSS and/or the terminals to be stopped.

To create a new user.jar file, issue the following command:

```
jar cfv user.jar *
```

To update an existing user.jar file and add the two new files, issue the following command:

```
jar uvf user.jar com/userco/retail/AEF/action/GiftCardActivationActionImpl.class userclass.properties
```

Executing the new action:
To perform the new action, the client creates an instance of ActionRequest, passing the class key and a HashMap of arguments to be used by the action. For this example, assume that the required arguments are a gift card number, and an amount. The client code to run the GiftCardActivationActionImpl is as follows:

```
// Assume cardNumber and cardAmount are String variables,
// and that automationProvider is a variable pointing to
// a valid POSAutomationProvider instance.
HashMap args = new HashMap();
args.put("cardNumber", cardNumber);
args.put("cardAmount", cardAmount);
// Create a new ActionRequest.  The class key for the action
// is the same key we have already put in userclass.properties
// to map to
// com.userco.retail.AEF.action.GiftCardActivationActionImpl.
ActionRequest request = new
             ActionRequest("GiftCardActivationAction", args);
Object result = null;
try
{
   result = automation.performAction(request);
}
```

```
catch (Throwable th)
{
   // Error processing here.
}
```

Service Provider

The SI Service Provider allows developers to provide additional value to the POS application via a messaging framework. The POS application publishes events and requests to an SI service bus that transports messages to and from subscribed services.

Developing services

Service developers must follow the guidelines below to connect their services to a POS application using an SI service provider. For each terminal, SI will instantiate one instance of any service registered with the AEF.

Define a service implementation:

- Create a module that implements SI's *com.ibm.retail.AEF.service.ServiceProvider* interface.
  - This module must implement the following methods:

public void startService(SIServiceBus serviceBus)	Called on the service to do whatever it needs to do to initialize. Services should register for topics during this part of its lifecycle. SI will assume that services can receive messages once they return from this method. **Note:** To prevent noticeably impacting startup, services should not execute long running tasks during this invocation.
public void stopService(SIServiceBus serviceBus)	Called as the AEF session is shutting down. Services should remove itself as topic listeners during this part of its lifecycle. **Note:** To prevent noticeably impacting shutdown, Services should not execute long running tasks during this invocation.
public void onEvent(String eventXML)	Process an event.
public String onRequest(String requestXML)	Handle a request and return an XML response.

- The service bus provides the following method for topic subscription:
  - public void subscribe(ServiceProvider yourService, List topics)
    - The first parameter will be the module SI invokes onEvent() and onRequest() methods on
    - The second parameter is a list of topic subscriptions. See the "Topics" on page 138 section in this document for information on the topics available to you.
- To unsubscribe, invoke the following method on the service bus:
  - Public void unsubscribe(ServiceProvider yourService)
    - This method unsubscribes the service from all topics.

Create a main jar file containing your service. Your jar file can also contain a class path entry in its MANIFEST.MF file for any additional jar files your service requires.

- Example:
  - Jar file name: foo_svc.jar
  - MANIFSET.INF
    - Class-path: foo_svc1.jar foo_svc2.jar

Topics

The following tables list the request topics available to a service.

Table 28. Request topics (1)

Topic name	Description
ProcessDigitalReceiptRequest	A topic on which SI will send you requests to process a digital receipt.

Table 29. Request topics (2)

Topic name	Description
CustomerEvent	Subscribe to this topic to receive an event when a the operator adds a customer loyalty identifier to the transaction.
ItemAddedEvent	Subscribe to this topic to receive an event when the operator adds an item to the transaction.
ItemVoidedEvent	Subscribe to this topic to receive an event when the operator voids an item.
ItemReturnedEvent	Subscribe to this topic to receive an event when the operator returns an item.
PSCouponAddedEvent	Subscribe to this topic to receive an event when a promotion service adds a coupon to the transaction.
PSCouponVoidedEvent	Subscribe to this topic to receive an event when a promotion service removes a coupon from the transaction.
PSGiftCardAddedEvent	Subscribe to this topic to receive an event when a promotion service adds a gift card to the transaction.
PSGiftCardVoidedEvent	Subscribe to this topic to receive an event when a promotion service removes a gift card from the transaction.
TenderAddedEvent	Subscribe to this topic to receive an event when the operator takes a tender.
TenderVoidedEvent	Subscribe to this topic to receive an event when the operator voids a tender.
TransactionStartEvent	Subscribe to this topic to receive an event when the operator starts a transaction.
TransactionCompletedEvent	Subscribe to this topic to receive an event when the operator completes a transaction.
TransactionVoidedEvent	Subscribe to this topic to receive an event when the operator voids a transaction.
TransactionSuspendedEvent	Subscribe to this topic to receive an event when the operator suspends a transaction.
TransactionRecoveredEvent	Subscribe to this topic to receive an event when the operator recovers a transaction.
TransactionRetrievedEvent	Subscribe to this topic to receive an event when the operator retrieves a transaction.
TransactionTransferredEvent	Subscribe to this topic to receive an event when the operator transferrers a transaction to another terminal.
TransactionReplayCompleteEvent	Subscribe to this topic to receive an event when the terminal completes a transaction recovery or retrieval.

Best practices

Below are best practices to follow:

- Your service must interact with resources in a thread safe manner. In the virtual session environment, the JVM could host multiple instances of your service (one per session) and you must be careful to avoid deadlocks in these environments.
- Because there could be multiple copies of your service running in the JVM, they should do things in a memory/ thread efficient way. On a 4690 controller, the JVM is limited to 320 threads, so avoid creating long running threads in your service. OS 4690 provides a thread pool, `com.ibm.retail.oscommon.util.ThreadPool`, in OS4690.zip to make thread management easier.
- Services are only allowed a limited amount of time to process a request and return a response. By default the limit is 10 seconds and can be changed using the si.service.timeout property in the services properties bundle. If a service doesn't respond before the time expires, SI will assume the service is hung and no longer communicate with it until the terminal restarts in order to prevent thread leaks and additional delays in terminal sales application response time. See the *Store Integrator User's Guide* for more information on the services properties bundle.

Digital receipts

Starting with SI Version 3 Release 4, SI can publish digital sales receipts in XML format to a service when used in conjunction with ACE Version 7 Release 4. A digital receipt XML document only contains information about the sales receipt. We do not provide a digital XML representation of gift receipts, lottery receipts, value card receipts, gift card receipts, non-sales portions of WIC transaction receipts, or other non-sales receipts.

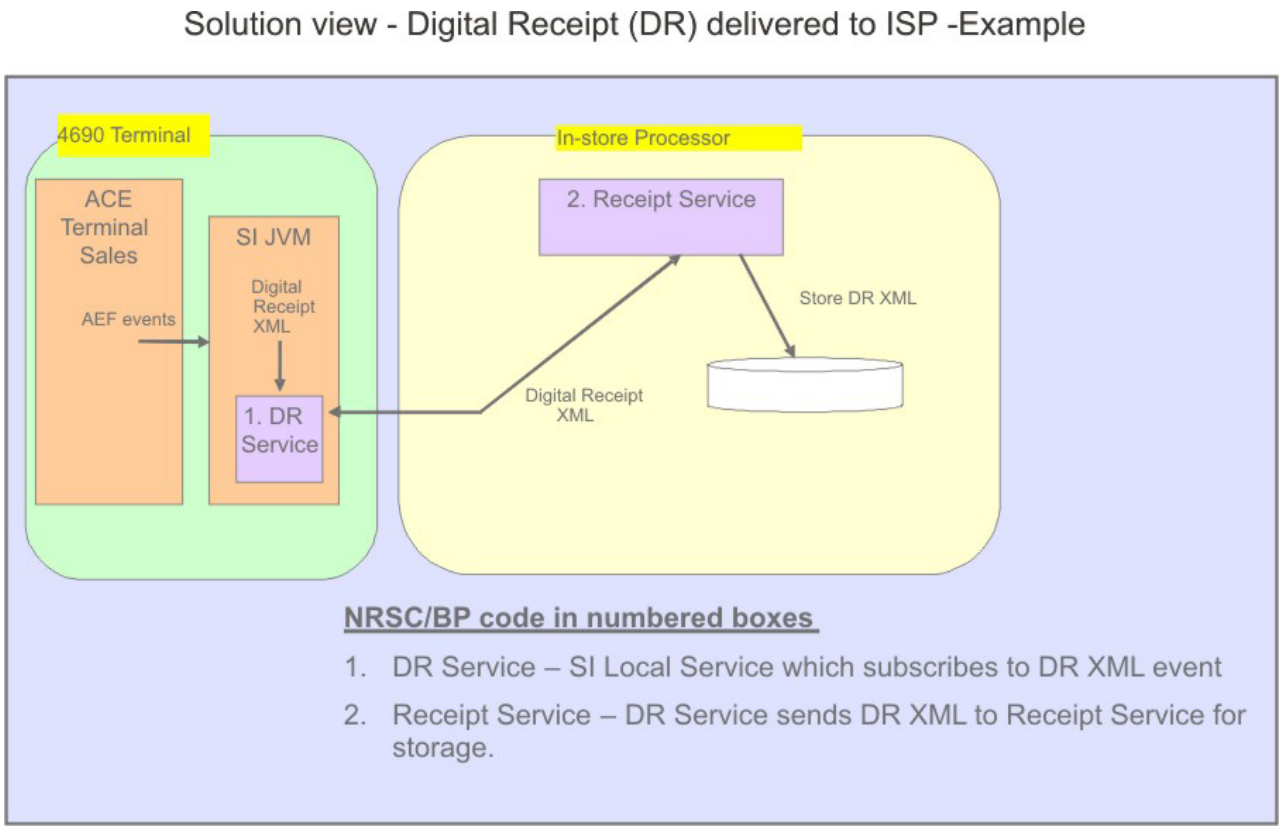
Solution view - Digital Receipt (DR) delivered to ISP -Example
4690 Terminal
ACE Terminal Sales
AEF events
SI JVM
Digital Receipt XML
1. DR Service
In-store Processor
2. Receipt Service
Store DR XML
Digital Receipt XML
NRSC/BP code in numbered boxes
1. DR Service – SI Local Service which subscribes to DR XML event
2. Receipt Service – DR Service sends DR XML to Receipt Service for storage.

Figure 63. Example Solution View

Figure 63 shows a digital receipt solution where the SI service initially receives the digital receipt XML document, sends it to an in-store processor who interrogates it, and then stores it.

If you wish to consume a digital receipt XML document, you must provide:

- The service in box #1 in Figure 63 on page 139 that initially receives the digital receipt XML document.
- A processor-like box #2 in Figure 63 on page 139 that will receive the XML document, process it, and send it to its next destination.

This document only describes the responsibilities of the service referenced in box #1. How the XML document is handled after that is defined by your solution architecture.

Payload

The digital receipt XML document contains two sections of data, one with formatted receipt image data and one with semantic item data.

- Formatted Image data
  - Enough information for the digital receipt XML document consumer to duplicate the 4610 version of the ACE formatted sales receipt (in html or pdf, for example)
  - Exceptions
    - Logos – we pass the name of the 4610 logos versus the binary logo data in the Digital Receipt XML document.
    - Signatures – ACE supports printing electronic signatures to the sales receipt. Signature data is not included in the XML document.
- Semantic data
  - Contains a subset of information about the transaction.
    - Customer information
    - Information on items and coupons sold
    - Information on the transaction totals and any tenders taken.
  - The semantic data section is intended for digital receipt XML document consumers who:
    - will apply their own formatting to the data
    - do not have a need to duplicate the original sales receipt

Whether or not SI will publish a digital receipt XML document and the payload contained within each document is controlled by ACE options. See the *SurePOS Application Client/Server Environment for 4690 OS: Planning and Installation Guide,* for information on the options related to digital receipts.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<rbi:ProcessDigitalReceiptRequest xmlns:rbi="http://business.toshiba.com/SI/RBI"
 MessageID="_gq5LYGk1EeKuLd7T5c2fIA">
  <Date>130128</Date>
  <Time>05:28</Time>
  <TransactionID>200</TransactionID>
  <StoreNumber>1</StoreNumber>
  <TerminalNumber>200</TerminalNumber>
  <OperatorID>200</OperatorID>
  <RequestID>216450005</RequestID>
  <DigitalReceipt>
    <ReceiptType>Customer</ReceiptType>
    <Logo>
      <URI>splogo</URI>
    </Logo>
    <TrailerText>
      <Text>TOTAL NUMBER OF ITEMS SOLD =    1  </Text>
      <Text>01/28/13 05:28am 1 200 200 200        </Text>
      <Text>                                      </Text>
      <Text>This order added 0.00  of coupons and </Text>
      <Text>200 points to bring your new totals   </Text>
      <Text>to 0.00  saved and 1900 points.       </Text>
      <Text>                                      </Text>
```

```
  <Text>We appreciate your loyal patronage    </Text>
  <Text>A B                                   </Text>
  <Text>  We Appreciate                       </Text>
  <Text> Customer Loyalty                     </Text>
  <Text>THANKS FOR SHOPPING WITH US TODAY     </Text>
  <Text>Terminal Number 200              </Text>
  <Text>Transaction Time 05:28am         </Text>
</TrailerText>
<ReceiptImage>
  <FormattedReceiptLine>
    <Feeds>0</Feeds>
    <Align>center</Align>
    <LineCategory>Header</LineCategory>
    <LineType>StoreHeader</LineType>
    <LogoReceiptLine>
      <URI>splogo</URI>
    </LogoReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Header</LineCategory>
    <LineType>StoreHeader</LineType>
    <TextReceiptLine>
      <Text>**************************************</Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Header</LineCategory>
    <LineType>StoreHeader</LineType>
    <TextReceiptLine>
      <Text>       The Mudd Store                 </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Header</LineCategory>
    <LineType>StoreHeader</LineType>
    <TextReceiptLine>
      <Text>       1593 Main Street               </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Header</LineCategory>
    <LineType>StoreHeader</LineType>
    <TextReceiptLine>
      <Text>       Waccaville, NC  27000          </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>PostPrintReceipt</LineCategory>
    <LineType>PostPrintReceipt</LineType>
    <TextReceiptLine>
      <Text> A B                       91234567890</Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
```

```
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>PostPrintReceipt</LineCategory>
  <LineType>PostPrintReceipt</LineType>
  <TextReceiptLine>
    <Text>        Item Number 1           2.00 B</Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>PostPrintReceipt</LineCategory>
  <LineType>PostPrintReceipt</LineType>
  <TextReceiptLine>
    <Text>        TAX                     0.20  </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>PostPrintReceipt</LineCategory>
  <LineType>PostPrintReceipt</LineType>
  <TextReceiptLine>
    <Text>   **** BALANCE                 2.20  </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>PostPrintReceipt</LineCategory>
  <LineType>PostPrintReceipt</LineType>
  <TextReceiptLine>
    <Text>        Cash                    2.20  </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>PostPrintReceipt</LineCategory>
  <LineType>PostPrintReceipt</LineType>
  <TextReceiptLine>
    <Text>        CHANGE                  0.00  </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>TotalItemCount</LineType>
  <TextReceiptLine>
    <Text>TOTAL NUMBER OF ITEMS SOLD =    1     </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>WorkstationInfo</LineType>
  <TextReceiptLine>
    <Text>01/28/13 05:28am 1 200 200 200        </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
```

```
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>                                        </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>This order added 0.00  of coupons and </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>200 points to bring your new totals   </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>2</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>to 0.00  saved and 1900 points.       </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>                                        </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>1</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>We appreciate your loyal patronage    </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
<FormattedReceiptLine>
  <Feeds>2</Feeds>
  <Align>center</Align>
  <LineCategory>Trailer</LineCategory>
  <LineType>LoyaltyMessages</LineType>
  <TextReceiptLine>
    <Text>A B                                     </Text>
  </TextReceiptLine>
</FormattedReceiptLine>
```

```
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>StoreTrailer</LineType>
    <BarcodeReceiptLine>
      <Height>162</Height>
      <Width>2</Width>
      <TextPosition>below</TextPosition>
      <Symbology>ITF</Symbology>
      <Value>00000120002001301280528</Value>
    </BarcodeReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>PumpTrailer</LineType>
    <TextReceiptLine>
      <Text>  We Appreciate                        </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>PumpTrailer</LineType>
    <TextReceiptLine>
      <Text> Customer Loyalty                     </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>PumpTrailer</LineType>
    <TextReceiptLine>
      <Text>THANKS FOR SHOPPING WITH US TODAY     </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>PumpTrailer</LineType>
    <TextReceiptLine>
      <HeightScale>1.5</HeightScale>
      <WidthScale>1.5</WidthScale>
      <Text>Terminal Number 200              </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
  <FormattedReceiptLine>
    <Feeds>1</Feeds>
    <Align>center</Align>
    <LineCategory>Trailer</LineCategory>
    <LineType>PumpTrailer</LineType>
    <TextReceiptLine>
      <HeightScale>1.5</HeightScale>
      <WidthScale>1.5</WidthScale>
      <Text>Transaction Time 05:28am         </Text>
    </TextReceiptLine>
  </FormattedReceiptLine>
</ReceiptImage>
```

```
    <RetailTransaction>
      <ItemCount>1</ItemCount>
      <CustomerInfo>
        <CustomerID>91234567890</CustomerID>
        <Name>A B</Name>
        <CustomerIDType>LOYALTY_NUMBER</CustomerIDType>
        <ReceivesDigitalReceipt>true</ReceivesDigitalReceipt>
        <PaperReceiptSuppressed>true</PaperReceiptSuppressed>
      </CustomerInfo>
      <LineItem>
        <ItemInfo>
          <Description>Item Number 1     </Description>
          <ItemIdentifier>1</ItemIdentifier>
          <ItemOrdinalNumber>1</ItemOrdinalNumber>
          <RegularUnitPrice>2.00</RegularUnitPrice>
          <ExtendedPrice>2.00</ExtendedPrice>
          <Weight>0.00</Weight>
          <DealPrice>0.00</DealPrice>
          <DealQuantity>1</DealQuantity>
          <DepartmentNumber>1</DepartmentNumber>
          <IsDiscountable>true</IsDiscountable>
          <PromotionCode>0</PromotionCode>
        </ItemInfo>
      </LineItem>
      <LineItem>
        <TenderInfo>
          <TenderType>Cash</TenderType>
          <TenderAmount>2.20</TenderAmount>
        </TenderInfo>
      </LineItem>
      <TotalInfo>
        <Total>2.20</Total>
        <SubTotal>2.00</SubTotal>
        <Tax>0.20</Tax>
      </TotalInfo>
     </RetailTransaction>
    <DigitalReceipt>
<rbi:ProcessDigitalReceiptRequest>
```

Creating a digital receipt service

To create a digital receipt service you must subscribe to the ProcessDigitalReceiptRequest when SI starts your service.

```
public void startService(SIServiceBus serviceBus)
{
   // Subscribe to receive the events stated in the configuration file
   if(serviceBus == null)
   {
      if(log.isWarnEnabled())
      {
         log.warn("Unable to subscribe to any messages. The bus is null.");
      }
   }
   else
   {
      ArrayList subscriptions = new ArrayList();
      // Add service required events
      subscriptions.add("ProcessDigitalReceiptRequest");
      log.trace("Subscribing to " + subscriptions.size() + " messages.");
      serviceBus.subscribe(this, subscriptions);
   }
   if(log.isInfoEnabled())
```

```
   {
      log.info("Digital receipt started");
   }
}
```

Once subscribed, SI will begin publishing digital receipt XML documents to your service via its onRequest() method. You must return a PublishDigitalReceiptResponse XML document indicating success or failure. Success means that the digital receipt XML document is guaranteed to reach its final destination. If your service returns a failure or doesn't respond at all, ACE will print a physical receipt for the customer.

```
public String onRequest(String requestXML)
{
   // do something with the digital receipt request
   ThreadPoolTask yourTask = new YourCustomTask(requestXML, yourOtherArgs...);
   ThreadPool.singleton().registerTask(this, 1, yourTask);
   ThreadPool.singleton().signal(this, 1);

   // here you would wait (< 10s to avoid SI timeouts) for the task to finish.
   return ProcessDigitalReceiptResponse XML document;
}
```

Example responses:

Success – An empty response indicates success to ACE.

```
<?xml version="1.0" encoding="UTF-8"?>
<rbi:ProcessDigitalReceiptResponse xmlns:
rbi="http://business.toshiba.com/SI/RBI"
MessageID="-2e174d02:138be145519:-2c13" CorrelationID="-2e174d02:138be145519:-2c14">
</rbi:ProcessDigitalReceiptResponse>
```

Failure – On a failure, you must populate the error element of the response. The presence of the Error element indicates a failure to ACE and it will print a receipt for the customer.

```
<?xml version="1.0" encoding="UTF-8?>
<rbi:ProcessDigitalReceiptResponse xmlns:rbi="http://business.toshiba.com/SI/RBI"
MessageID="-2e174d02:138be145519:-2c13" CorrelationID="-2e174d02:138be145519:-2c14">
 <Error>
  <Message>Some error occured.</Message>
  <ErrorCode>123</ErrorCode>
 </Error>
</rbi:ProcessDigitalReceiptResponse>
```

Programming with the POSBC

POSBC clients send XML request messages and receive back XML response messages. The following is an example of a Java client sending requests to and receiving responses from POSBC.

Create a JIF socket using the IP address of the machine the POSBC is running on. The port id will default to 6697, unless another value is specified in the POSBC configuration properties file.

```
Socket aSocket =  new SISocketFactory("JIF").createSocket("10.10.1.10", 6697);
```

Get an input stream and an output stream for the socket.

```
DataInputStream anInputStream = new DataInputStream(aSocket.getInputStream());
DataOutputStream anOutputStream = new DataOutputStream(aSocket.getOutputStream());
```

Create an initialize request and set the session number that you plan to use. In this case, the initial state requested is SIGNED_ON.

```
InitializeRequestType initRequest = SchemaFactory.INSTANCE.createInitializeRequestType();
initRequest.setTerminalNumber( aSession );
initRequest.setInitialState
```

To find all of the requests that are available for the POSBC, reference the POSBC_requests.xsd schema. The required and optional input parameters for each request are contained in that schema. Here is an example of the InitializeRequestType entry:

```
<xsd:complexType name="InitializeRequestType">
 <xsd:complexContent>
  <xsd:extension base="BasicRequestType">
   <xsd:sequence>
    <xsd:element name="TerminalNumber" type="xsd:string"/>
    <xsd:element name="Recovery" type="xsd:boolean" minOccurs="0"/>
   <xsd:element name="SessionServerURI"type="xsd:string" minOccurs="0"/>
    <xsd:element name="InitialState"type="InitialStateType" minOccurs="0"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

The TerminalNumber element is required, and the Recovery, SessionServerURI, and InitialState elements are optional.

Create an initialize type object and set the initialize request field to the initialize request object that was created.

```
InitializeType initObj = SchemaFactory.INSTANCE.createInitializeType();
initObj.setInitializeRequest( initRequest );
```

Create an XML string from the initialize object with UTF-8 format.

```
String request = new String(JavaXMLConverter.javaToXML((EObject)initObj), "UTF-8");
```

Format the request header and the XML request string. Each request header must contain a session id that indicates to the POSBC which instance it needs to use to process that request. Typically, the request ID is “1”, unless the POSBC is connected to multiple sessions.

```
byte[] msg;
MessageHeaders requestHeader = new MessageHeaders();
requestHeader.addHeader(BCConstants.SESSION_ID, theSessionID);
requestHeader.addHeader(BCConstants.MESSAGE_TYPE, BCConstants.REQ_MSG);
msg = format(requestHeader, request.getBytes("UTF-8"));
```

Send the message out on the output stream.

```
anOutputStream.writeInt(msg.length);
anOutputStream.write(msg);
anOutputStream.flush()
```

Loop waiting for a result or an event on the socket input stream. If it is an event, send the event to all the event listeners.

```
String result = null;
boolean loopReply = true;
while (loopReply)
{
   int bufLength = anInputStream.readInt();
   byte[] buffer = new byte[bufLength];
   anInputStream.readFully(buffer);
   result = new String(buffer, "UTF-8");
   MessageHeaders headersType = new MessageHeaders();
   SoepsProtocol.getInstance().parse(buffer, headersType);

   // check for a response or an event
   String header =
```

```
   headersType.getHeaderField(BCConstants.MESSAGE_TYPE);
   if (header.equals(BCConstants.RESP_MSG))
   {
      loopReply = false;
   }
   else if (headersType.getHeaderField(
   BCConstants.MESSAGE_TYPE).equals(BCConstants.EVENT_MSG))
   {
      byte[] event = stripSoapHeader(result).getBytes("UTF-8"));
      if (event != null) {
         for (int i = 0; i < posbcStatusEventListeners.size();
              i++)
      {
         (posbcStatusEventListeners.get(i)).onEvent(event);
      }
   }
}
```

You can end the POSBC instance at any time, regardless of its state.

Detecting that a device is in use

When the POSBC detects a JavaPOS device that it needs to use is being used by another application, it sends the following status event to its clients.

```
<schema:POSBCStatusEvent xmlns:schema="http://bc.si.retail.ibm.com/POSBCSchema">
 <POSStatus>
  <Severity>WARNING</Severity>
  <Status>JPOS_DEVICE_IN_USE</Status>
  <StatusMessage>An external application is performing an operation preventing JavaPOS devices
       from being used at this time. Application name: [application]</StatusMessage>
 </POSStatus>
</schema:POSBCStatusEvent>
```

POSBC Clients can only receive this event during an Initialize request. Once the POSBC sends the event, it waits while the AEF's DeviceInUseNotificator attempts to open the device. Once the device opens, the POSBC continues with its initialization.

The client will only receive this event once per Initialization. To receive another event, the client must send the POSBC a Terminate request followed by another Initialize request. Alternatively, the POSBC will send the client another event if it disconnects while the POSBC is waiting for the device to open.

Client code written in C connecting with the POSBC

The following is a price checking client for the POSBC written in C language. This client connects to a socket on the localhost, 127.0.0.1, at port 6697, the default port the POSBC listens on. The client then asks for a terminal number and uses the input to initialize the POSBC. Once the POSBC is initialized, the client asks the user to input an item code which is used to create the add item request that is sent to the POSBC. The POSBC returns an add item response to the client who parses it and extracts the unit price. The client then prints the unit price to the screen and voids the transaction started by the add item request. The client is ended by entering 'q' at the item code prompt. All events from the POSBC are discarded by the client.

```
 /*
 * Simple Pricer
 * Nov 30, 2006
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#include <windows.h>
#include <winsock.h>
#include <io.h>
#include <sys/types.h>


// settestenv.bat
// to compile: icc posbctest.c wsock32.lib
// runBC - run posbc

// POSBC XML shema information
char schemaInfo[54] = " xmlns:xsd2=\"http://bc.si.retail.ibm.com/POSBCSchema\"";

// The following variables are used to create the initialize request XML.
char initTag[17] = "xsd2:Initialize";
char initReq[18] = "InitializeRequest";
char termNum[15] = "TerminalNumber";

// The following variables are used to create the add item request XML.
char addItemTag[14] = "xsd2:AddItem";
char addItmReq[15] = "AddItemRequest";
char itemIdent[15] = "ItemIdentifier";
char keyedItemID[12] = "KeyedItemID";

// The following variables are used to create the void transaction request XML.
char voidTrxTag[22] = "xsd2:VoidTransaction";
char voidTrxReq[23] = "VoidTransactionRequest";

char theRequest[65536]; // string to hold the request. Maximum size is 65536
char theResponse[65536]; // string to hold the response.

int port = 6697; // the port to connect the socket to.
char *theHostName = "localhost"; // the host to connect to.
int theSocket=0; // the socket number. This will be obtained when creating the socket.

/* function prototypes */
long initializeSocket(void);
void addElementBegin(char *data, char *msg);
void addElementEnd(char *data, char *msg);
void addBegin(char *msg);
void addEnd(char *msg);
long handleResponse(void);
long sendRequest(const char *mesg);
long initialize(char *terminalNumber);
long addItem(char *itemID);
long voidTransaction(void);


/* ------------------------------------------------------ *
 *  main()
 * ------------------------------------------------------
```

Figure 64. Connecting code example (Part 1 of 8)

```
void main()
{
    long rc=0;
    int  retVal=0;
    char inputData[13];
    char itemID[13];
    char terminalNumber[3];

 rc = initializeSocket();

    // initialize the POSBC
    printf("Enter the terminal number:\n");
    if (gets(inputData) != NULL) {
        terminalNumber[0] = '\0';
        retVal = sscanf(inputData, "%s", terminalNumber);
        rc = initialize(terminalNumber);
    }

    while (rc >= 0)
    {
        printf("\nEnter item ID to be priced ('q' to quit).\n");
        if (gets(inputData) != NULL) {
           itemID[0] = '\0';
           retVal = sscanf(inputData, "%s", itemID);
           if (strcmp(itemID, "q") == 0) {
               rc = -1;
               continue;
           }
           // start a transaction by adding an item
           rc = addItem(itemID);
           if (rc < 0) {
               // no reason to void since the add resulted in an error
               rc = 0;
               continue;
           }
           // void the transaction transaction
           rc = voidTransaction();
        }
    }
    close(theSocket);
}
/* ------------------------ End of main() ------------------ */
```

Figure 65. Connecting code example (Part 2 of 8)

```
//*****************************************************************************
// initializeSocket()
// - Creates and connects to the socket used for POSBC communication.
//
// Returns: A positive number if the socket is created successfully.
//*****************************************************************************
long initializeSocket(void)
{
    short  rc;
    struct sockaddr_in server;  // server information
    struct hostent *hostEntry;  // host name information

    // startup the windows socket interface using version 1.1
    WSADATA wsaData;
    WORD version = MAKEWORD(1,0);
    WSAStartup(version, &swaData);

    // get host information.
    hostEntry = gethostbyname(theHostName);
    if (((char *)hostEntry) == NULL) {
       printf("Error getting host!\n");
       return -1;
    }

    // clear the memory area for the server information.
    memset((char *)&server,0,sizeof(server));

    // Put the server information into the server structure.
    server.sin_family = AF_INET;
    server.sin_port   = htons(port); // specify the port in network byte order
    memcpy(&server.sin_addr, hostEntry->h_addr, hostEntry->h_length);

    // Create a stream socket
    if ((theSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
       printf("Error creating socket.\n");
       return -1;
    }

    // Connect to the server
    rc=connect(theSocket, (struct sockaddr *)&server, sizeof(server));
    if(rc < 0) {
       printf("Error connecting to server.\n");
       close(theSocket);
    }
    return rc;
}
```

Figure 66. Connecting code example (Part 3 of 8)

```
//*****************************************************************************
// initialize(char *terminalNumber)
// - Initializes the POSBC.
//
// Parameters: char *terminalNumber
//             - The terminal number to use.
// Returns: A positive number if the POSBC is initialized successfully.
//*****************************************************************************
long initialize(char *terminalNumber)
{
    long rc = 0;
    char req[65536];
    memset(req, '\0', sizeof(req));

    addBegin(req);
    strcat(req, initTag);
    strcat(req, schemaInfo);
    addEnd(req);
    addElementBegin(initReq, req);

    addElementBegin(termNum, req);
    strcat(req, terminalNumber);
    addElementEnd(termNum, req);

    addElementEnd(initReq, req);
    addElementEnd(initTag, req);

    printf("Initializing the POSBC...\n");
    rc = sendRequest(req);
    if (rc > 0) {
        printf("POSBC initialized.\n");
    }

   return rc;
}

//*****************************************************************************
// addItem(char *itemID)
// - Sends an add item transaction request to the POSBC.
//
// Parameters: char *itemID
//             - The item to add to the transaction.
// Returns: A positive number if the item is added successfully.
//*****************************************************************************

long addItem(char *itemID)
{
    long rc = 0;
    char req[65536];
    char* ptr;
    char* price;
    int loop = 0;

    memset(req, '\0', sizeof(req));

    printf("Adding item %s.\n", itemID);
    addBegin(req);
    strcat(req, addItemTag);
    strcat(req, schemaInfo);
    addEnd(req);
    addElementBegin(addItmReq, req);
    addElementBegin(itemIdent, req);
    addElementBegin(keyedItemID, req);
    strcat(req, itemID);
    addElementEnd(keyedItemID, req);
    addElementEnd(itemIdent, req);
    addElementEnd(addItmReq, req);
    addElementEnd(addItemTag, req);
```


Figure 67. Connecting code example (Part 4 of 8)

```
 rc = sendRequest(req);
    if (rc >=0) {
        // get the price from the response.
        ptr = strstr(theResponse, "<RegularUnitPrice>");
        if (ptr != NULL) {
            // move the ptr past the end of the opening tag
            ptr += strlen("RegularUnitPrice>");
            printf("The unit price of item %s is <", itemID);
            // print all characters up until the begining of the closing tag.
            while (*ptr != '<') {
                printf("%c", *ptr);
                ptr += 1; // move to the next character.
            }
        }
        printf(".\n");
    }
    return rc;
}

//*****************************************************************************
// voidTransaction()
// - Sends a void transaction request to the POSBC.
//
// Returns: A positive number if the transaction is voided successfully.
//*****************************************************************************
long voidTransaction(void)
{
    long rc = 0;
    char req[65536];
    memset(req, '\0', sizeof(req));

    printf("Voiding the transaction.\n");
    addBegin(req);
    strcat(req, voidTrxTag);
    strcat(req, schemaInfo);
    addEnd(req);
    addElementBegin(voidTrxReq, req);
    addElementEnd(voidTrxReq, req);
    addElementEnd(voidTrxTag, req);

    rc = sendRequest(req);
    return rc;
}

//*****************************************************************************
// addElementBegin(char *data, char *msg)
// - Appends the data to the message, enclosing the data in XML tag opening
//   braces (< and >). Example: if the data is 'hello', the string <hello>
//   will be appended to the message.
//
// Parameters:  char *data
//              - The element to append to the string.
//              char *mesg
//              - the string onto which the element will be appended.
//*****************************************************************************
```

Figure 68. Connecting code example (Part 5 of 8)

```
void addElementBegin(char *data, char *msg)
{
    strcat(msg, "<");
    strcat(msg, data);
    strcat(msg, ">");
}

//*****************************************************************************
// addElementEnd(char *data, char *msg)
// - Appends the data to the message, enclosing the data in XML tag closing
//   braces (</ and >). Example: if the data is 'hello', the string </hello>
//   will be appended to the message.
//
// Parameters:  char *data
//              - The element to append to the string.
//              char *mesg
//              - the string onto which the element will be appended.
//*****************************************************************************
void addElementEnd(char *data, char *msg)
{
    strcat(msg, "</");
    strcat(msg, data);
    strcat(msg, ">");
}

//*****************************************************************************
// addBegin(char *msg)
// - Appends the DIF message header, complete with the opening XML brace, to
//   the input string. The header conains the text
//   'soeps~Session-Id=1|Message-Type=REQ|~<'.
//
// Parameters:  char *mesg
//              - the string to append the message header to.
//*****************************************************************************
void addBegin(char *msg)
{
    strcat(msg, "soeps~Session-Id=1|Message-Type=REQ|~<");
}

//*****************************************************************************
// addEnd(char *msg)
// - Appends a closing brace, '>', to a string. This method is used to close
//   XML tags.
//
// Parameters:  char *mesg
//              - the string to add the closing brace to.
//*****************************************************************************
void addEnd(char *msg)
{
    strcat(msg, ">");
}
```

Figure 69. Connecting code example (Part 6 of 8)

```
//*****************************************************************************
// sendRequest(const char *req)
// - Sends the message passed in to the POSBC using the
//   already created socket and then waits for a response.
//   If the response contains an exception result, the error
//   code is printed and -1 is returned.
//
// Parameters:  const char* req
//               - The request to send to the POSBC.
// Returns: A positive number if the message is sent successfully and the
//          response is received successfully.
//*****************************************************************************
long sendRequest(const char *req)
{
    int rc = 1; // a response code to validate the send
    int loop = 0; // loop counter
    int length = strlen(req); // the length of the message to send

    // add the length of the message as the first four bytes in
    // reverse order.
    for(loop=3;loop>=0;loop--) {
        theRequest[loop]=(char)length;
        length >>= 8;
    }

    // add 4 here for two reasons, one to correctly control the loop,
    // and two because later it's used to send the entire message that
    // contains the 4 extra bytes indicating the length.
    length = strlen(req)+4;
    for (loop=4; loop;lenght; loop++) {
        theRequest[loop] = req[loop-4];
    }

    rc = send(theSocket, theRequest, length, 0);
    if (rc < 0) {
        printf("Failed to send message = %s\n", theRequest);
        rc = -1;
    }
    memset(theRequest,'\0', length); // clean up the memory area
    return handleResponse();
}

//*****************************************************************************
// handleResponse()
// - Reads an incoming message from the socket. Event type messages are ignored.
//   When a response is received, it is checked for exceptions before the
//   method ends. If an exception is found, the error code is printed and -1
//   is returned.
//
//  Returns: A positive number if the response was successfully received and
//           does not contain any exceptions.
//*****************************************************************************
char *ptr; // pointer to memory location for the response
    char *msgType; // the type of message we read
    char msgLength[4]; // array to hold the length of the message
    int amountRead = 0; // integer for comparisons when reading messages
    int length = 0; // the length of the data
    int loop = 0; // control variable
    int rc = 1; // response code for the reads
```

Figure 70. Connecting code example (Part 7 of 8)

```
    // loop to capture any events and the response.
    do {
        // first get the length of the response.
        rc = recv(theSocket, msgLength, 4, 0);
        if (rc < 0) {
            printf("Failed to get the message size from the socket. rc= %d.\n", rc);
            break;
        }
        for(loop=0; loop < 4; loop++) {
            length=msgLength[loop] | (length << 8);
        }

        // make sure the memory space for the response is actually empty
        memset(theResponse, '\0', sizeof(theResponse));

        // ptr points to the place in memory we should read the data into
        // this moves as we read data.
        ptr = theResponse;
        amountRead = 0;
        while(amountRead < length && rc > 0) {
            // loop until we read all of the message
            rc = recv(theSocket, ptr, length-amountRead, 0);
            if (rc < 0) {
                printf("Failed to read the data from the socket. rc= %d.\n", rc);
                return rc;
            }
            else {
                ptr += rc; // move the pointer
                amountRead += rc; // keep track of the amount of data read
            }
        }
        // get the message type
        msgType = strstr(theResponse, "Message-Type=RESP");
        if (msgType != NULL) {
            // check the response for exceptions
            ptr = strstr(theResponse, "<ExceptionResult>");
            if (ptr != NULL) {
                printf("An exception occurred processing the request:\n");
                // find the error code
                ptr = strstr(theResponse, "<ErrorCode>");
                if (ptr != NULL) {
                    // move the ptr past the end of the opening tag
                    ptr += strlen("<ErrorCode>");
                    printf("Error Code= ");
                    // print all characters up until the begining of the closing tag.
                    while (*ptr != '<') {
                        printf("%c", *ptr);
                        ptr += 1; // move to the next character.
                    }
                }
                printf("\n");
                return -1;
            }
        }
    } while(msgType == NULL); // msgType is null when the msg is not a response
    return rc; // should be the response code from the last read.
}
```

Figure 71. Connecting code example (Part 8 of 8)

GS1 DataBar and the POSBC

The GS1 DataBar specification permits GS1 DataBar expanded barcodes to contain the special characters <, >, &, and 0x1d (an unprintable grouping separator). Version 1.0 of the XML specification, however,

does not permit any of these special characters. Therefore, POSBC clients supporting GS1 DataBar barcodes must escape these characters using the generally accepted escape sequences of <, >, & ^], respectively. When sending XML responses back to the client, the POSBC will also escape these special characters in the same way.

Coding guidelines

Using error logging

Error logs are critical in detecting and correcting application errors. The more is known about the application's activity at the time of the problem, the easier it is to resolve. Unfortunately, logging too much information can have a negative impact on performance. Therefore, finding the right amount of information to log is very important. This task shows you how to log errors in a way that is consistent with the rest of the Store Integrator code. Topics such as error log entry contents, APIs, and correct use of logging levels are discussed.

Logging levels

Use logging levels consistently to avoid the negative impact of logging too much system information. Consistent logging enables you to know the type of information that is logged at each level. You can set the logging level for your system to provide just the information that you need. Table 30 shows the appropriate use for each logging level.

Table 30. Error logging levels

Level	Description
FATAL	Severe errors that cause premature termination. Expect these to be immediately visible on a status console.
ERROR	Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console. This level is used for: • Java exceptions • Missing data files such as EALAEFXF.DAT or EAMAEFXF.DAT • Missing property files
WARN	Use of deprecated APIs, poor use of API, *almost* errors, other runtime situations that are undesirable or unexpected, but not necessarily wrong. Expect these to be immediately visible on a status console. This level is used for: • Missing bundle keys or values. • Factory cannot create object messages. • Invalid values. • Condition lock time-outs that include condition states.
INFO	Interesting runtime events (startup/shutdown). Expect these to be immediately visible on a console, so be conservative and keep to a minimum. This level is used for: • Error Helper creation. • Error Helper handleError() return values. • AEFBase configuration. • Dump of the Bundles that includes all the values from the property files. • Java Health Pipe creation.
DEBUG	Detailed information about the flow through the system. Expect these to be written to logs only. This level is used for: • Automation call summary (for example, Operator 999999 logon called) • SalesTranscaction call summary (for example, Add item 3 (itemid toString())called) • XML statements received from base application

Table 30. Error logging levels (continued)

Level	Description
TRACE	More detailed information. Expect these to be written to logs only. By default the message priority should be no lower than INFO. That is, by default debug and trace messages should not be seen in the logs. It is recommended that you have exception or problem information available for first-pass problem determination in a production level enterprise application without turning on DEBUG or TRACE as default log levels. Too much information is available in these levels to be appropriate for day-to-day operations. This level is used for method entry and exit tracing.

Logging APIs

Table 31 lists the APIs and their effect after the Jakarta Commons and Sun JDK 1.4 differences are factored. These are the APIs that Store Integrator uses. Jakarta Commons Logging Wrapper enables you to do implementation independent logging. So you should treat log.fatal() and log.error() differently even though they look the same in the error logs. That way if your logging implementation changes to one that distinguishes between error and fatal levels, your software will take advantage of that.

Table 31. Log levels with sample log entries

API	Log Level	Sample Log Entry
Log.fatal() **Log.error()**	SEVERE	Apr 21, 2003 8:10:48 AM Test main SEVERE: error text Apr 21, 2003 8:10:48 AM Test main SEVERE: error text
Log.warn()	WARNING	Apr 21, 2003 8:10:48 AM Test main WARNING: error text
Log.info()	INFO	Apr 21, 2003 8:10:48 AM Test main INFO: error text
Log.debug()	FINE	Apr 21, 2003 8:10:48 AM Test main FINE: error text
Log.trace()	FINEST	Apr 21, 2003 8:10:48 AM Test main FINEST: error text

Controlling error log output

The output of the logging calls is configurable based on the settings of the logging.properties file specified when the JVM was started. All of the Store Integrator response files include an entry for a logging properties file to use (-Djava.util.logging.config.file=ADX_IPGM:cmaxlogs.pro). Refer to the IBM Store Integrator User's Guide for more information about controlling the output of the logging calls. The end user has the ability to control what information makes it into the error logs. Since logging an error involves some *overhead*, it is recommended that a test be done before an error is logged. This way if the error log is just going to be thrown away, the *overhead* is avoided. Specifically, it is recommended that log.isInfoEnabled(), log.isDebugEnabled(), and log.isTraceEnabled() be called before logging an error at those levels. For example:

```
if log.isInfoEnabled()
{
   log.info("Text to be logged");
}
```

Error log entry contents

There are two versions of the log APIs for each logging level. One that takes a string and one the takes a string and an exception as a parameter. To standardize the log entries, it is recommended that any errors that include exceptions, should be logged using the two parameter version of the API. Converting the exception objects into a string leads to inconsistent looking log entries. To further standardize the log entries, Store Integrator provides a class called AEFMessage. This class has two attributes. The first is the message that contains the text part of the error. The second is sessionID that should be set to the terminal number of the terminal logging the error. These attributes should be set and this class should be passed as the string parameter of the logging calls. The following code is an example of logging using the AEFMessage class:

```
msg = new AEFMessage();
if (log.isDebugEnabled())
{
   msg.setSessionID(terminalNumber);
   msg.setMessage("Creating ApplicationDataConnector.");
   log.debug(msg);
}
```

Sample error logging program

The following program shows how the logging code should be used. The first three lines are the import statements needed for AEFMessage and the logging classes. In line 6 the Log object is created. It is recommended that the Log object have the same name as the class. So in this case we pass Test.class to the getLog method. On line 9 a new AEFMessage object is created. The terminal number for this AEFMessage object is initialized on line 10. Lines 11 through 15 are used to insure that the logging calls are not made if this level of logging is not enabled for this log object (commonly referred to as a code guard). Line 13 initializes the message attribute of the AEFMessage object. In line 14 a trace message is logged. Lines 16 through 20 are the same as 11 through 15 except that in this case a debug message is logged. Lines 21 through 33 show how to log warning and fatal messages. As you can see no code guard was used for these calls because the errors are higher than informational. Lines 34 through 38 log a trace message to match the one from lines 11 through 15.

```
1. import com.ibm.retail.AEF.util.*;
2. import org.apache.commons.logging.Log;
3. import org.apache.commons.logging.LogFactory;
4. public class Test
5. {
6.   private static Log log = LogFactory.getLog(Test.class);
7.   public static void main(String[] args) throws Exception
8.   {
9.     AEFMessage msg = new AEFMessage();
10.     msg.setSessionID("001");
11.     if (log.isTraceEnabled())
12.     {
13.        msg.setMessage("Entering Test.main().");
14.        log.trace(msg);
15.      }
16.      if (log.isDebugEnabled())
17.      {
18.        msg.setMessage("Test.main() called with the following arguments: " + args);
19.        log.debug(msg);
20.      }
21.      try
22.      {
23.        if (args.length == 0)
24.        {
25.          msg.setMessage("Main called with no arguments.  Defaults will be used.");
26.          log.warn(msg);
27.        }
28.      }
29.      catch (Exception e)
30.      {
31.        msg.setMessage("An exception was thrown in the main method of Test.");
32.        log.fatal(msg, e);
33.      }
34.      if (log.isTraceEnabled())
35.      {
36.        msg.setMessage("Leaving Test.main().");
37.        log.trace(msg);
38.      }
39.    }
40.  }
```

Figure 72. Sample error logging program

Writing an error helper

An error helper is a class that is called by the AEFErrorHandler class to handle errors or provide input to the base application. The idea is for each error helper to handle a specific error or input sequence. If the

error or condition is corrected, the error helper returns a value of ErrorHelper.ERROR_HANDLED to the AEFErrorHandler. This includes a case where the original error or condition is cleared but another error is detected. If the error or condition is not corrected, then the error helper returns ErrorHelper.ERROR_IS_FATAL to the AEFErrorHandler.

Error helper classes

All the error helpers must implement the ErrorHelper interface. Store Integrator provides several abstract implementations of this interface. These include ErrorHelperImpl, SAErrorHelper, ACEErrorHelper, and GSAErrorHelper. Store Integrator also provides a default error helper that gets called if an error is encountered that has no error helper configured for it. This default error helper is DefaultErrorHelper. This error helper always returns ErrorHelper.ERROR_IS_FATAL and sets its outer exception to indicate that the default error helper was used.

ErrorHelper interface methods

The following methods are specified in the ErrorHelper interface.

getError
Used to get the AEFError object that contains the information about the error that this helper is trying to handle. ErrorHelperImpl provides an implementation of this method.

setError
Sets the error that this helper is working on. ErrorHelperImpl provides an implementation of this method.

getOuterException
Returns the outer most Exception (if any) that was thrown while handling this error. ErrorHelperImpl provides an implementation of this method.

setOuterException
Sets the outer most Exception that was thrown while handling this error. ErrorHelperImpl provides an implementation of this method.

getSleepInterval
Returns the number of milliseconds to wait before continuing with the next operation while handling errors. During testing, it was found that differences in hardware, and software configuration could lead to timing issues. In many cases, invalid key sequence errors were erroneously generated. To compensate for this, Java sleep calls were added to the AEFErrorHandler and ErrorHelper classes. The Sleep Interval is configured in config.properties. All of these timing issues have been corrected but this method was left here in case it is needed in the future. ErrorHelperImpl provides an implementation of this method.

sleep Causes the Error Helper to wait before continuing with the next operation. During testing, we found that differences in hardware, and software configuration could lead to timing issues. In many cases, invalid key sequence errors were erroneously generated. To compensate for this, sleep calls (using this method) were added to the AEFErrorHandler and ErrorHelper classes. The default sleep interval is configured in config.properties. All of these timing issues have been corrected but this method was left here in case it is needed in the future. The default value for the sleep interval was set to -1 to avoid the call to the Thread.sleep method. ErrorHelperImpl provides an implementation of this method.

isEmulatedDevice
Used to determine if a device is real or emulated. ErrorHelperImpl provides an implementation of this method.

getCurrentState
Gets the current state of the base application. ErrorHelperImpl provides an implementation of this method.

setInputDevicesLocked
Locks the input devices. ErrorHelperImpl provides an implementation of this method.

resetInputDevices
Reset the input devices to the state prior to the setInputDevicesLocked call. ErrorHelperImpl provides an implementation of this method.

getErrorHandler
Gets the AEFErrorHandler that created this error helper. ErrorHelperImpl provides an implementation of this method.

setErrorHandler
Sets the AEFErrorHandler that created this error helper. ErrorHelperImpl provides an implementation of this method.

handleError
This method is the one that does the work specific to clearing this error. Each error helper must implement this method.

Error helper bundles

The error and class bundles provide information that the AEF error handler needs to create the error helpers. From the error bundle, the error helper gets commands:

1. A text description of the error.
2. The key (from the class bundle) of the error helper to use in handling the error. (If a class name is provided, the system creates that class, but it is better to provide a key instead).
3. The error code associated with this error. This code is extracted from the AEFConst.java file. If a number is used instead of a code, the error helper uses the number instead. Decimal numbers from 9000 to 9999, and -9000 to -9999 are reserved for user programming.
4. The extended error code associated with this error. This code is extracted from the AEFConst.java file. If instead of a code a number is used, the error helper uses that number instead. Decimal numbers from 9000 to 9999, and -9000 to -9999 are reserved for user programming.

   From the class bundle, the error helper gets the class to use to create the error helper. The correct place to make changes to these bundles is the userclass.properties, and usererror.properties files. These files should be included in one of the user JAR files so that the application has access to them at runtime.

Sample informational error helper

The following code shows a simple error helper. This is the SA error helper that is used for informational error messages. These are messages where the operator can acknowledge the message but is not given any other choices. Lines 1 through 11 are the import statements needed for a typical error helper. On line 13 the class declaration extends SAErrorHelper. SAErrorHelper extends ErrorHelperImpl so most of the methods of the ErrorHelper interface have been implemented already. In lines 33 through 117 the handleError method is implemented. Three error handling modes are available. Of these, only two, automatic (POSAutomationProvider.HANDLE_AUTOMATIC), and default (POSAutomationProvider.HANDLE_DEFAULT) are supported. The automatic mode is used to let Store Integrator attempt to clear the errors without operator intervention. This is helpful for customer facing devices where the consumer should not be asked to clear errors or enter manager override codes, and so on. In the default mode, Store Integrator waits for the operator to clear the error or input the data, then continues with the action. This is useful for devices that are operated by store personnel.

```
1.   package com.ibm.retail.AEF.util;
2.   import com.ibm.retail.AEF.action.*;
3.   import com.ibm.retail.AEF.automation.*;
4.   import com.ibm.retail.si.util.*;
5.   import com.ibm.retail.si.Copyright;
6.   import com.ibm.retail.AEF.thread.*;
7.   import com.ibm.retail.AEF.session.*;
8.   import java.rmi.*;
9.
10.  import org.apache.commons.logging.Log;
11.  import org.apache.commons.logging.LogFactory;
12.
13.  public class SAErrorHelperInformational extends SAErrorHelper
14.  {
15.
16.     // Constructor
17.     public SAErrorHelperInformational() throws AEFException
18.     {
19.     }
20.
21.     /**
22.      * Tries to resolve the error condition if possible.
23.      * @param lock The ConditionLock that returned the error.
24.      * @param keySequenceAction The key sequence that generated the error.
25.      * @param goodConditions An array of the conditions that would indicate
                                success after the key sequence is sent.
26.      * @param badConditions An array of the conditions that would indicate
                               a failure after the key sequence is sent.
27.      * @param timeout An integer used to determine how long to wait for one
                         of the listed conditions before a timeout exception
                         is thrown.
28.      * @param satisfiedCondition The index of the condition in the badConditions
                                    array that was satisfied (needs to be adjusted
                                    as it is a negative number).
29.      * @return int Returns ErrorHelper.ERROR_HANDLED if the error was cleared
                      or dealt with. Returns ErrorHelper.ERROR_IS_FATAL if
30.      *             the error was not cleared or dealt with.
31.      */
32.
33.     public int handleError(ConditionLock lock, AEFAction keySequenceAction,
          Condition[] goodConditions, Condition[] badConditions, int timeout, int satisfiedCondition)
34.     {
35.        int retVal = 0;
36.        int handlingMode = 0;
37.        try
38.        {
39.           handlingMode = getErrorHandlingMode();
40.           switch(handlingMode)
```

Figure 73. Simple Error Helper (Part 1 of 3)

```
41.          {
42.             case POSAutomationProvider.HANDLE_AUTOMATIC:
43.             sleep(); // Added to resolve timing issues. See sleep method for more info.
44.             retVal = clearError();
45.             break;
46.             case POSAutomationProvider.HANDLE_DEFAULT:
47.             // Unlock keyboard if need be so that the operator can interact.
48.             setInputDevicesLocked(false);
49.             retVal = waitForOperatorToClear();
50.             break;
51.             case POSAutomationProvider.HANDLE_CALLBACK:
52.             // Not Implemented for SA in this release!
53.             default:
54.             throw new AEFException(AEFConst.UNSUPPORTED_ERROR_HANDLING_MODE,
                                       AEFConst.NONE, "An Error Handling Mode of " +
                                       handlingMode + " was returned which is not valid.");
55.          }
56.       }
57.       catch (RemoteException re)
58.       {
59.          AEFException tmpException = new AEFException(AEFConst.ERROR_HANDLER_EXCEPTION_DETECTED,
                    AEFConst.RMI_REMOTE_EXCEPTION, "SAErrorHelperInformational.handleError(): Remote
        Exception was detected.", re);
60.          tmpException.appendExceptions(getOuterException());
61.          setOuterException(tmpException);
62.          retVal = ErrorHelper.ERROR_IS_FATAL;
63.          if (log.isWarnEnabled())
64.          {
65.             tempAEFMessage.setMessage("There was a remote exception thrown in handleError of
        SAErrorHelperInformational for error "
                                          + getError().getErrorKey()+ ".");
66.             log.warn(tempAEFMessage, re);
67.          }
68.       }
69.       catch (AEFException e)
70.       {
71.          e.appendExceptions(getOuterException());
72.          setOuterException(e);
73.          retVal = ErrorHelper.ERROR_IS_FATAL;
74.          if (log.isWarnEnabled())
75.          {
76.             tempAEFMessage.setMessage("There was an AEF exception thrown in handleError of
       SAErrorHelperInformational for error "
                                          + getError().getErrorKey()+ ".");
77.             log.warn(tempAEFMessage, e);
78.          }
79.       }
80.       finally
```

Figure 74. Simple Error Helper (Part 2 of 3)

```
81.        {
82.           // Reset the input devices.
83.           try
84.           {
85.              resetInputDevices();
86.           }
87.           catch (AEFException e)
88.           {
89.              e.appendExceptions(getOuterException());
90.              setOuterException(e);
91.              retVal = ErrorHelper.ERROR_IS_FATAL;
92.              if (log.isWarnEnabled())
93.              {
94.                 tempAEFMessage.setMessage("There was an AEF exception thrown in handleError of
       SAErrorHelperInformational for error "
                                          + getError().getErrorKey()+ ".");
95.                 log.warn(tempAEFMessage, e);
96.              }
97.           }
98.        }
99.       if (retVal < 0)
100.       {
101.          retVal = ErrorHelper.ERROR_IS_FATAL;
102.       }
103.       else
104.       {
105.          retVal = ErrorHelper.ERROR_HANDLED;
106.       }
107.       if (log.isDebugEnabled())
108.       {
109.          tempAEFMessage.setMessage("Leaving handleError for error " +
                                       getError().getErrorKey()+ ".");
110.          log.debug(tempAEFMessage);
111.          tempAEFMessage.setMessage("The return code was " + retVal + ".");
112.          log.debug(tempAEFMessage);
113.          tempAEFMessage.setMessage("The Error Handling mode used was " +
                                       handlingMode + ".");
114.          log.debug(tempAEFMessage);
115.       }
116.       return retVal;
117.    }
118.
119.    // Instance Variables
122.    private static Log log = LogFactory.getLog(SAErrorHelperInformational.class);
121. }
```

Figure 75. Simple Error Helper (Part 3 of 3)

Writing your own error helper

This task is illustrated using the SA application.

In this task you write an error helper that notifies store personnel when a B082 PICKUP NEEDED SOON prompt is displayed by a terminal. By default B082 errors in the Supermarket application are handled by the SAErrorHelperInformational class (shown in the example above). So in automatic mode the error is cleared by the system. In default mode the error is cleared by the operator. In neither case does the system log an error. If the error helper above was modified to log an error, then the *Remote Management* capabilities built into Store Integrator allow you to monitor for this error and take preventive actions. (Refer to the *Remote Management Agent and Viewer User's Guide*, GC30-4106 for more information about monitoring errors.)

Perform the following steps to accomplish this task:

1. Write the new error helper.
2. Determine what error codes to use.
3. Modify the Error bundle.

4. Modify the Class bundle.
5. Include all the new and modified files in a user JAR file.
6. Install the user.jar file on the system.

Write the new error helper

The following code shows the new error helper. This java file name and path should be d:\si\user\com\userco\retail\AEF\util\PickupNeededSoonErrorHelper.java. This is the SA error helper (discussed above) that is used for informational error messages with a few new lines added. The new lines (41 through 45) are used to log an error message.

```
1.   package com.userco.retail.AEF.util;
2.   import com.ibm.retail.AEF.action.*;
3.   import com.ibm.retail.AEF.automation.*;
4.   import com.ibm.retail.si.util.*;
5.   import com.ibm.retail.si.Copyright;
6.   import com.ibm.retail.AEF.thread.*;
7.   import com.ibm.retail.AEF.session.*;
8.   import com.ibm.retail.AEF.util.*;
9.   import java.rmi.*;
10.
11.  import org.apache.commons.logging.Log;
12.  import org.apache.commons.logging.LogFactory;
13.
14.  public class PickupNeededSoonErrorHelper extends SAErrorHelper
15.  {
16.
17.     // Constructor
18.     public PickupNeededSoonErrorHelper() throws AEFException
19.     {
20.     }
21.
22.     /**
23.      * Tries to resolve the error condition if possible.
24.      * @param lock The ConditionLock that returned the error.
25.      * @param keySequenceAction The key sequence that generated the error.
26.      * @param goodConditions An array of the conditions that would indicate
                                success after the key sequence is sent.
27.      * @param badConditions An array of the conditions that would indicate
                               a failure after the key sequence is sent.
28.      * @param timeout An integer used to determine how long to wait for one
                         of the listed conditions before a timeout exception is
                         thrown.
29.      * @param satisfiedCondition The index of the condition in the badConditions
                                    array that was satisfied (needs to be adjusted
                                    as it is a negative number).
30.      * @return int Returns ErrorHelper.ERROR_HANDLED if the error was cleared
                    or dealt with. Returns ErrorHelper.ERROR_IS_FATAL if the
31.      *          error was not cleared or dealt with.
32.      */
33.
34.     public int handleError(ConditionLock lock, AEFAction keySequenceAction,
35.     Condition[] goodConditions, Condition[] badConditions, int timeout, int satisfiedCondition)
36.     {
37.        int retVal = 0;
38.        int handlingMode = 0;
39.        try
40.        {
41.           handlingMode = getErrorHandlingMode();
42.           if (log.isInfoEnabled())
43.           {
44.              tempAEFMessage.setMessage("Pickup Needed Soon!");
45.              log.info(tempAEFMessage);
46.           }
47.           switch(handlingMode)
48.           {
49.              case POSAutomationProvider.HANDLE_AUTOMATIC:
50.                 sleep(); // Added to resolve timing issues. See sleep method for more info.
51.                 retVal = clearError();
52.                 break;
```

```
53.              case POSAutomationProvider.HANDLE_DEFAULT:
54.                 // Unlock keyboard if need be so that the operator can interact.
55.                 setInputDevicesLocked(false);
56.                 retVal = waitForOperatorToClear();
57.                 break;
58.              case POSAutomationProvider.HANDLE_CALLBACK:
59.              // Not Implemented for SA in this release!
60.              default:
61.                 throw new AEFException(AEFConst.UNSUPPORTED_ERROR_HANDLING_MODE,
                           AEFConst.NONE, "An Error Handling Mode of " + handlingMode
                           + " was returned which is not valid.");
62.           }
63.        }
64.        catch (AEFException e)
65.        {
66.           e.appendExceptions(getOuterException());
67.           setOuterException(e);
68.           retVal = ErrorHelper.ERROR_IS_FATAL;
69.           if (log.isWarnEnabled())
70.           {
71.              tempAEFMessage.setMessage("There was an AEF exception thrown in handleError "
                                    + "of PickupNeededSoonErrorHelper for error "
                                    + getError().getErrorKey()+ ".");
72.              log.warn(tempAEFMessage, e);
73.           }
74.        }
75.        finally
76.        {
77.           // Reset the input devices.
78.           try
79.           {
80.              resetInputDevices();
81.           }
82.           catch (AEFException e)
83.           {
84.              e.appendExceptions(getOuterException());
85.              setOuterException(e);
86.              retVal = ErrorHelper.ERROR_IS_FATAL;
87.              if (log.isWarnEnabled())
88.              {
89.                 tempAEFMessage.setMessage("There was an AEF exception thrown in handleError "
                                       + "of PickupNeededSoonErrorHelper for error "
                                       + getError().getErrorKey()+ ".");
90.                 log.warn(tempAEFMessage, e);
91.              }
92.           }
93.        }
94.        if (retVal < 0)
95.        {
96.           retVal = ErrorHelper.ERROR_IS_FATAL;
97.        }
98.        else
99.        {
100.          retVal = ErrorHelper.ERROR_HANDLED;
101.       }
102.       if (log.isDebugEnabled())
103.       {
104.          tempAEFMessage.setMessage("Leaving handleError for error "
                                    + getError().getErrorKey()+ ".");
105.          log.debug(tempAEFMessage);
106.          tempAEFMessage.setMessage("The return code was " + retVal + ".");
107.          log.debug(tempAEFMessage);
108.          tempAEFMessage.setMessage("The Error Handling mode used was "
                                    + handlingMode + ".");
109.          log.debug(tempAEFMessage);
110.       }
111.       return retVal;
112.    }
113.
114.    // Instance Variables
115.    private static Log log = LogFactory.getLog(PickupNeededSoonErrorHelper.class);
116. }
```

The destination directory is c:/si/userjar. Make sure that this directory exists, then compile the file by executing the following command:

```
javac -classpath c:/si/si_jars/aefsa.jar;c:/si/si_jars/aefsys.jar;
c:/si/si_jars/siutil.jar -d c:/si/userjar
com/userco/retail/AEF/util/PickupNeededSoonErrorHelper.java
```

This new class must be compiled and the PickupNeededSoonErrorHelper.class from this step will be included in the user.jar file. See "Include all new and modified files in a user jar file."

Determine what error codes to use

In the AEFConst.java file, there is a group of public static final integer variables that should be used to represent the error codes associated with a specific error. There is also a group of public static final integer variables that should be used to represent the *extended* error codes associated with a specific error. Sometimes you can to reuse these for your errors. Other times, you must use one of the numbers reserved for user errors. In this case you create a new error code and reuse an extended error code. Use 9000 for the error code. Use the variable PICKUP_NEEDED for the extended error code. These values are used in the following error bundle description.

Modify the error bundle

You must either create a new c:/si/userjar/usererror.properties file or modify an existing one. When you have the file, add or modify the lines for error B082 as needed to match the lines below.

```
# B082 Pickup Needed Soon
B082_ADDITIONAL_TEXT=The first cash drawer limit has been exceeded please do a pickup.
B082_HANDLING_CLASS_KEY= PickupNeededSoonErrorHelper
B082_ERROR_CODE=9000
B082_EXTENDED_ERROR_CODE= PICKUP_NEEDED
```

Modify the class bundle

You must either create a new c:/si/userjar/userclass.properties file or modify an existing one. When you have the file, add the following line to the file:

```
PickupNeededSoonErrorHelper=com.userco.retail.AEF.util.PickupNeededSoonErrorHelper
```

Include all new and modified files in a user jar file

This example shows three modified or created files. These files are c:/si/userjar/usererror.properties, c:/si/userjar/userclass.properties, and c:/si/userjar/com/userco/retail/AEF/util/PickupNeededSoonErrorHelper.class (from PickupNeededSoonErrorHelper.java). To create a new user.jar file that includes all of these files, invoke the following command from a userjar directory:

```
jar cvf user.jar *
```

To add the files to an existing file, invoke the following command from the userjar directory:

```
jar uvf user.jar *
```

Install the user.jar file on the system

The response files that are provided with Store Integrator include d:\user\user.jar in the classpath. So the new user.jar should be put into that directory and distributed to all the controllers. If JavaTOF is being used with SI GUI, then the d:\user\user.jar must be added to ADX_IPGM:SUREVIEW.DUJ, and ADX_IPGM:SUREVIEW.RCI. ADX_IPGM:TOF.BAT must then be run. If bundling is being used, rebuild any terminal preload files that need to be rebuilt by running ADXPLDRB.386. After the terminals are reloaded, the new error handler is invoked when the B082 error is encountered.

Handling exceptions

When Java encounters an error during program execution, it throws an exception. These exceptions are a subclass of the Java Throwable class. The Store Integrator API is designed to throw AEFException and RemoteException objects. The following sections provide more detailed information about these types of exceptions and how to deal with them.

AEFExceptions

AEFException extends the standard Java Exception class. This class is used to add additional information to exceptions that is unique to Store Integrator.

AEFException has the following protected attributes.

- int errorCode (Defaults to 0. See AEFConst.java for more information.)
- int errorCodeExtended (Defaults to 0. See AEFConst.java for more information.)
- String errorKey (Defaults to null)
- Throwable originalThrowable;
- AEFException previousException (Defaults to null)
- AEFException nextException (Defaults to null)
- AEFError errorObject (Defaults to null)

AEFException has the following public methods.

1. int getErrorCode() (Returns the error code for the exception.)
2. void setErrorCode(int newErrorCode) (Sets the error code for this exception.)
3. int getErrorCodeExtended() (Returns the extended error code for the exception.)
4. void setExtendedErrorCode(int newErrorCode) (Sets the extended error code for this exception.)
5. String getDescription() (Returns the description for the exception.)
6. String getErrorKey() (Returns the AEFErrorKey for this exception.)
7. void setErrorKey(String newErrorKey) (Sets the AEFErrorKey for this exception.)
8. AEFException getPreviousException() (Gets the previous AEFException on the list of chained exceptions.)
9. void setPreviousException(AEFException prevException) (Sets the previous AEFException on the list of chained exceptions.)
10. AEFException getNextException() (Gets the next AEFException on the list of chained exceptions.)
11. void setNextException(AEFException newNextException) (Set the next AEFException on the list of chained exceptions.)
12. Throwable getOriginalThrowable() (Gets the original throwable (the cause in JVM 1.4 >) for this exception.)
13. void setOriginalThrowable(Throwable newOriginalThrowable) (Set the original throwable (the cause in JVM 1.4 >) for this exception.)
14. void appendExceptions(AEFException exceptionsToAppend) (This method takes an AEFException and sets the Previous Exception attribute for the last AEFException object on this list (the object that called the method) to point to it. The idea is to have this (the object that called the method) object (the newest exception) be at the start of the list of exceptions. Traversing the list of previous exception attributes leads you back in time until the oldest exception in the chain is reached.)
15. void printExceptions() (This method prints all the exceptions in the chain starting with this exception.)
16. String getStackText(Throwable currentThrowable, int indentLevel) (This method returns the stack trace for a throwable as a String.)
17. String getAEFExceptionText(AEFException currentAEFException, int indentLevel) (This method returns the information for this AEFException as a String.)
18. String toString() (This method prints a summary of the exceptions.)
19. AEFException getOriginalException() (Gets the original AEFException (last in the chain) on the list of chained exceptions.)
20. void setErrorObject(AEFError newErrorObject) (Sets the AEF error object for this exception.)
21. AEFError getErrorObject() (Gets the AEF error object for this exception.)

As noted above, AEFException has a previousException and a nextException attribute. This allows Store Integrator to chain these exceptions together in the case where more than one AEFException was generated. These exceptions are chained together with the oldest one at the end of the list. So the AEFException returned by any SI call is the exception that was most recently generated. If the intent of the handling code is just to log the exception, then the AEFException returned by the call should be logged. The error logging code uses the toString method of AEFException to correctly format all the exceptions on the list. If the code is trying to determine the original problem so as to take corrective action, then the original exception should be examined. To get the original (oldest) exception you would have to call the getOriginalException method.

When the original AEFException has been retrieved, the getErrorCode, getErrorCodeExtended, getErrorKey, and getOriginalThrowable methods should be called to try to determine how to handle the error. The getErrorCode method always returns a value from AEFConst.java. The getErrorCodeExtended method returns a value from AEFConst.java or zero if there is no extended code for this error. The getErrorKey method returns the key for this error if applicable. These keys are listed in the error bundle but for the most part they match the base application error codes (A003 or B007, and so on). If the return value from these methods is zero or null, then more than likely a Java exception caused the error. The getOriginalThrowable method should be called to determine the cause of the error.

Java exceptions

As with any java program, Store Integrator programs need to be coded to handle exceptions correctly. Aside from any error recovery code, it is also very important to log errors whenever an exception is detected. Note that unchecked exceptions do not require explicit try/catch blocks. So if all the error logging is done inside catch statements for checked exception, some very important messages could be missed. To avoid this, it is recommended that the main() method of your application has a catch (throwable) block that logs any exception that it catches.

Timeout considerations

Some of the most difficult exceptions to recover from are timeout exceptions. These are AEFException objects with an error code of AEFConst.OPERATION_TIMEOUT or any Java exception that has the word timeout in its name. The most common is the SocketTimeoutException as a result of an RMI call. The following code is a sample stack trace:

```
java.rmi.UnmarshalException: Error unmarshaling return header; nested exception is:
        java.net.SocketTimeoutException: Read timed out
        at sun.rmi.transport.StreamRemoteCall.executeCall(Unknown Source)




at sun.rmi.server.UnicastRef.invoke(Unknown Source)
at com.ibm.retail.AEF.automation.SalesTransactionImpl_Stub.addTender(Sales
TransactionImpl_Stub.java:385)

at
sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)

at java.lang.reflect.Method.invoke(Unknown Source)

at org.mozilla.javascript.NativeJavaMethod.call(Unknown Source)

at org.mozilla.javascript.ScriptRuntime.call(Unknown Source)

at org.mozilla.javascript.Interpreter.interpret(Unknown Source)
at org.mozilla.javascript.InterpretedScript.call(Unknown Source)
at
org.mozilla.javascript.InterpretedScript.exec(Unknown Source)
```

```
at
org.mozilla.javascript.Context.evaluateReader(Unknown Source)
at
org.mozilla.javascript.Context.evaluateString(Unknown Source)
at
com.ibm.bsf.engines.javascript.JavaScriptEngine.eval(JavaScriptEngine.java:83)
at scripter.AEFScript.processLine(AEFScript.java:765)
at scripter.AEFScript.run(AEFScript.java:387)
Caused by: java.net.SocketTimeoutException: Read timed out
at java.net.SocketInputStream.socketRead0(Native Method)
at java.net.SocketInputStream.read(Unknown Source)
at java.io.BufferedInputStream.fill(Unknown Source)
at java.io.BufferedInputStream.read(Unknown Source)
at java.io.DataInputStream.readByte(Unknown Source)
... 17 more
```

The cause for most timeouts is either an enablement error or a slow system. In the case of an enablement error, a Store Integrator automation call is waiting for a condition that is never satisfied. So after the appropriate timeout period, the exception is thrown. To correct this problem, either code or property files must be modified. Code changes are required if the condition list is incorrect. That is the automation call did not anticipate a possible outcome to the automation call. Property file changes are needed if a function code, state, or sub-state value is incorrect in a property file.

A special consideration must be given if the AEF error handling mode is set to *default*. Because the application waits for the operator, there is a chance that any one of the various calls made by the application will timeout. Therefore, it is recommended that if the default error handling mode is used that the value of any property in the config.properties file that ends with read.timeout be changed to 0. This causes the read calls to wait indefinitely, allowing the operator time to take the necessary action. In the case of a slow system, the timeout values can be increased. The only danger is that if the cause of the timeout is really an enablement error, the operator will have to wait that much longer for the exception.

The reason that the timeout exceptions are difficult to process is that when the exception is thrown, the Store Integrator call and the base application disconnect. The Store Integrator code throws the exception that works its way up to the caller. The base application in the case of a slow system eventually runs the automation call and gets to the correct state. So after the timeout, the Store Integrator application tries to resolve the outcome of the command that threw the timeout exception. How this is done varies from one automation call to the next. But the basic idea is to keep track of how many items, tenders, and other activities had been added to the current transaction. State and substate information can also be helpful. Monitoring the base application after the timeout error for any of these changes can give you an indication that the automation call has been processed.

Tender recovery

When a client application tries to add a tender to a transaction, many things can go wrong. Such things as power failures, network problems, or other problems can make it difficult to determine whether or not the command succeeded. Also an exception or a timeout may be received because the system is behind on processing requests. Unfortunately, only so much can be done with the program to determine the real status of the tender request. Here are some suggestions of things to do to try to determine if the tender went through.

Before adding a tender, the client code should get information about the transaction that includes the transaction number, the number of tenders, transaction totals, all pertinent tender information. This data must be logged to persistent storage.

If a client receives either a RemoteException or an AEFException while trying to add a credit tender to the transaction, the tender operation is considered *in-flight* on the POS application. Perform the following steps to determine whether or not the credit went through with approval.

1. After the error, use the POSAutomationProvider.getTransaction method to see if a transaction is in progress. If no transaction is in progress, then the tender was taken and the transaction is completed. (Tender satisfied the balance due.) If a transaction is in progress, then continue with the next step.
2. If a transaction is in progress, then call the Transaction.getNumberOfTenders and Transaction.getTransactionTotals methods and compare the results to the numbers from step one. If these numbers changed, then the tender was accepted. Otherwise, continue with the next step.
3. At this point, the status of the request to add the tender is not known. Log an error to request that someone will use the tools provided by the base application to determine the status of the tender request.

Sample code

The following code shows how to catch an AEFException (line 110), determine what the error codes are (lines 112-119), and recover from the error (lines 123-147). It also shows how to log the throwable objects as described above to make sure that unchecked exceptions are logged (lines 209-214).

```
1.   /*
2.    * AEFExample2
3.    *
4.    * 07/14/04
5.    *
6.    * Copyright:
7.    * Licensed Materials - Property of IBM
8.    * "Restricted Materials of IBM"
9.    * 5724-AEF
10.   * (C) Copyright IBM Corp. 2004, 2007.
12.   */
13.  package com.userco.retail.client.example2;
14.
15.  import com.ibm.retail.si.util.*;
16.  import com.ibm.retail.AEF.server.*;
17.  import com.ibm.retail.AEF.session.*;
18.  import com.ibm.retail.AEF.automation.*;
19.  import com.ibm.retail.AEF.factory.*;
20.
21.  import java.rmi.*;
22.  import java.util.*;
23.
24.  import org.apache.commons.logging.Log;
25.  import org.apache.commons.logging.LogFactory;
26.
27.  public class AEFExample2
28.  {
29.
30.     /*
31.      * The main routine
32.      *
33.      * @param args an array of strings for command line parameters
34.      */
35.     public static void main(String[] args)
36.     {
37.        String errorText = null;
38.
39.        System.out.println("Starting AEFExample2...");
40.        try
41.        {
42.           SessionServer server = AEFBase.getInstance().getSessionServer();
43.           if (server != null)
44.           {
45.              //connected
46.              System.out.println("Obtained SessionServer");
47.              AEFSession session = server.getAvailableSession();
```

```
48.              if (session != null)
49.              {
50.                 System.out.println("Obtained session. Terminal number is " +
51.                                    session.getTerminalNumber());
52.
53.                 System.out.println("Get an automation provider");
54.                 POSAutomationProvider automation =
55.                 session.getPOSAutomationProvider();
56.
57.                 // wait until the session is ready for commands
58.                 System.out.println("Wait until session is ready for commands...");
59.                 boolean sessionReady = session.waitUntilReady(10000);
60.                 int retryCount = 0;
61.                 while (!sessionReady)
62.                 {
63.                    retryCount++;
64.                    System.out.println("Retrying... #" + retryCount);
65.                    sessionReady = session.waitUntilReady(10000);
66.                    if (retryCount > 5)
67.                    {
68.                       break;
69.                    }
70.                 }
71.                 if (sessionReady)
72.                 {
73.                    System.out.println("Session is ready for commands.");
74.
75.                    // initialize
76.                    System.out.println("Initialize the session to signon on state.");
77.                    automation.initialize(POSAutomationProvider.SIGNON);
78.
79.                    // logon with default - set op/pw in userlogon.properties on 4690
80.                    System.out.println("Logon using default logon.");
81.                    Operator op = automation.logon();
82.                    System.out.println("Logged on as " + op.getInfo().getID());
83.
84.                    // start a new transaction
85.                    System.out.println("Start a new transaction.");
86.                    SalesTransaction t1 = automation.startTransaction();
87.                    System.out.println("Transaction" +
88.                         t1.getTransactionInfo().getTransactionID() + " started.");
89.
90.                    // add item 33 to the transaction
91.
92.                    try
93.                    {
94.                       System.out.println("Add an item.");
95.                       ArrayList itemList = t1.addItem("33");
96.                       if ((itemList != null) && (!itemList.isEmpty()))
97.                       {
98.                          Iterator it = itemList.iterator();
99.                          while (it.hasNext())
100.                         {
101.                            Item item = (Item)(it.next());
102.                            System.out.println("Item added. Description = " +
103.                                               item.getInfo().getDescription());
104.                         }
105.
106.                         System.out.println("This is transaction " +
107.                                            t1.getTransactionInfo().getTransactionID() + ".");
108.                      }
109.                   }
110.                   catch (AEFException aie)
```

```
111.                  {
112.                     // Get the oldest exception in the list.
113.
114.                     AEFException originalException = aie.getOriginalException();
115.
116.                     // Get the error codes.
117.
118.                     int errorCode = originalException.getErrorCode();
119.                     int extendedErrorCode = originalException.getErrorCodeExtended();
120.
121.                     // See if we can get around this error.
122.
123.                     if (errorCode == AEFConst.INVALID_ARGUMENT && extendedErrorCode ==
                             AEFConst.ITEM_PRICE_REQUIRED)
124.                     {
125.                        String tempPrice = getPrice();
126.
127.                        ItemIdentifier itemID = (ItemIdentifier)(FactoryImpl.createObject
          (ItemIdentifier.CLASS_KEY));
128.                        itemID.setItemCode("33");
129.                        itemID.setItemCodeType(AEFConst.VELOCITY);
130.                        itemID.setPrice(tempPrice);
131.
132.                        // Try to add the item again.
133.
134.                        System.out.println("Add an item (second try).");
135.                        ArrayList itemList = t1.addItem(itemID);
136.                        if ((itemList != null) && (!itemList.isEmpty()))
137.                        {
138.                           Iterator it = itemList.iterator();
139.                           while (it.hasNext())
140.                           {
141.                              Item item =(Item)(it.next());
142.                              System.out.println("Item added.  Description = " +
143.                                                 item.getInfo().getDescription());
144.                           }
145.
146.                           System.out.println("This is transaction " +
147.                              t1.getTransactionInfo().getTransactionID() + ".");
148.                        }
149.                     }
150.                  }
151.
152.                  // void transaction
153.                  System.out.println("Void transaction.");
154.                  t1.voidTransaction();
155.
156.                  // logoff this operator
157.                  System.out.println("Logoff.");
158.                  automation.logoff();
159.
160.                  // release to make the session available again
161.                  System.out.println("Release the session.");
162.                  session.release();
163.                  session=null;
164.               }
165.               else
166.               {
167.                  errorText = "Timed out waiting for session ready";
168.                  System.err.println(errorText);
169.                  if (log.isWarnEnabled())
170.                  {
171.                     log.warn(errorText);
```

```
                  }
               }
            }
            else
            {
               errorText = "Unable to obtain AEFSession.  ";
               System.err.println(errorText);
               if (log.isWarnEnabled())
               {
                  log.warn(errorText);
               }
            }
         }
         else
         {
            // couldn't connect -- URL is incorrect or server is down

             errorText = "Unable to connect to AEF SessionServer.  ";
             System.err.println(errorText);
             if (log.isWarnEnabled())
             {
                log.warn(errorText);
             }
          }
       }
       catch (AEFException ae)
       {
          errorText = "An AEF exception occurred: ";
          System.err.println(errorText + ae.getMessage());
          log.error(errorText, ae);
       }
       catch (RemoteException re)
       {
          errorText = "A remote exception occurred: ";
          System.err.println(errorText + re.getMessage());
          log.error(errorText, re);
       }
       catch (Throwable e)
       {
          errorText = "A throwable object was caught: ";
          System.err.println(errorText + e.getMessage());
          log.error(errorText, e);
       }
       System.out.println("AEFExample2 complete.");
       System.exit(0);
    }

    /**
     * Gets the item's price.
     *
     * @return String The item's price.
     */
    public static String getPrice()
    {
       // We will assume that there is a way to get the price from some other source.
       // The point of this example is to show how to check and handle the error.
       // Not how to get the price.

       return "12.00";
    }

    private static Log log = LogFactory.getLog(AEFExample2.class);
}
```

Building, debugging and packaging

Using properties file configuration

The behavior of AEF can be changed by modifying the values in various properties files. This task shows you how to change the behavior of all the terminals, or a subset of terminals. This subset can include real terminals, a specific terminal, or a group of terminals that belong to a session role.

Properties files bundles

The AEF bundles are sets of properties files that are used to configure AEF. Each bundle has a starting file that includes a link to the next file in the bundle (nextFile=). Each file is read in sequential order. If the value of a property is specified in multiple files, then the last value read is used. This method allows users to override AEF default values. AEF includes the following bundles:

Note: For a complete list of properties file names, see the AEF bundles table in the *Store Integrator User's Guide*, (SC30-4085).

- Automation
- Class
- Configuration
- Error
- Function Code
- Internationalization
- Key Sequence
- Logon
- Services
- Session
- SSL
- State
- Substate
- Tender Map

  All the AEF base properties files are included in the AEF.JAR file. The APP*.properties files are in the application specific JAR files (AEFGSA.JAR, AEFSA.JAR, or AEFACE.JAR). Any USER*.properties files must also be included in a JAR file. Put your user property overrides into one of the user JAR files. For a complete list of JAR files, see the table in the *Store Integrator User's Guide*, (SC30-4085).

For a complete list of the properties in each file, see Appendix D, "AEF property files," on page 203 or use the commands described below to extract the files from the JAR files.

Manipulating JAR files

A JAR file is a Zip compatible file that contains Java class files and other resources (like images and properties files). Since these are Zip files, any Zip file utility can be used to manipulate them. If you do not have access to any Zip utilities, the JDK comes with a **jar** command.

Table 32 lists some commands for working with JAR files.

Table 32. JAR commands

Function	Command
Create a new jar file containing user properties.	jar cvf user.jar *
List the contents of the aef.jar file.	jar tvf aef.jar

For more information about the JAR command, see the documentation that came with your JDK or type jar on the command line to get a list of all the command line options.

Changing the default user ID and password for all terminals

The Automation Provider includes a logon ID that uses default operator ID and passwords. When the POSAutomationProvider.logon method is called without arguments, the terminal number is used to lookup an operator id and password. The Logon bundle defines the default operator ID and password for each terminal number, and is made-up of the applogon.properties, and userlogon.properties files. The applogon.properties file that is provided on the SI CDs contains the following lines:

```
#termno=id,password
001=1,1
002=1,1
100=1,1
999=2,2
nextFile=
```

The first line (#termno=id,password) is a comment line. Any line that starts with a # is considered to be a comment. Lines 2 through 4 specify that terminals 1, 2, and 100 should use operator ID 1 and a password of 1, if the logon method is called without any arguments. Line 5 specifies that terminal 999 should use ID 2, and a password of 2, if the logon method is called without any arguments. The last line specifies the next file in the bundle.

To change these values, you need to create a file called c:/si/userjar/userlogon.properties. Since this is typically the last file in the chain, it does not need a nextFile line. In this example use the terminal number as the ID and password. So the file would look like this.

```
# Terminal 1 user 1 password 1
001=1,1
# Terminal 2 user 2 password 2
002=2,2
# Terminal 3 user 3 password 3
003=3,3
# Terminal 4 user 4 password 4
004=4,4
```

Now that you have a userlogon.properties file, you need to include it in a JAR file that is in the classpath. Because you want this change to affect all the terminals, you add this file to the user.jar file. To create a new user.jar file that includes the userlogon.properties file, switch into the c:/si/userjar directory, and use the command

jar cvf user.jar *

If you have already created a user.jar file, use the command

jar uvf user.jar userlogon.properties

The resulting user.jar file now needs to be copied to the c:\user directory on the 4690 controller (including setting file distribution attributes). In the case of TOF terminals, the TOF.BAT file must be rerun, or if bundling is being used, rebuild any terminal preload files that need to be rebuilt by running ADXPLDRB.386. Then reload the terminals.

Changing the session trace level of a specific terminal

Both virtual and real sessions write trace information into a session trace buffer that can be used for troubleshooting. In this task you set the session trace level for a specific terminal session. This could be helpful if you are experiencing a problem that requires more log information to troubleshoot than what is normally generated by the system. You could set all the terminals to do full logging but this could hurt performance. By setting the level of one terminal, you get the additional data and do not degrade the performance of the other terminals.

The session trace level is controlled by the following line in the session.properties file:

trace.level=COARSE

To override this value for only one terminal (terminal 100 for example) you need to create a file called c:/si/userjar/usersession_100.properties. This file contains only the following line:

trace.level=FINE

When you have created this file, you need to get it into a JAR file that is in the classpath. To create a new user.jar file that includes the usersession_100.properties file, use the following command:

```
jar cvf user.jar *
```

To add the usersession_100.properties file to an existing user.jar file, use the following command:

```
jar uvf user.jar usersession_100.properties
```

The resulting user.jar file now needs to be copied to the c:\user directory on the 4690 controller (including setting file distribution attributes). If terminal 100 uses TOF, the TOF.BAT file must be rerun, or if bundling is being used, rebuild any terminal preload files that need to be rebuilt by running ADXPLDRB.386. Then reload the terminals.

Changing the session trace buffer size for terminals by role

In this task you set the session trace buffer size for a group of terminals that share the same session role. A role is a logical group of terminal sessions. In this example, you create a role for all the fuel pump terminal sessions. The session trace buffer size is controlled by the following line in the session.properties file:

```
trace.buffer.size=1000
```

To override this value for only the fuel pump terminals (terminals 200, 201, and 300 to 310 for example) you must create two new files. One is c:/si/userjar/usersessionrole.properties. This file is used to define the role and its terminals. The file contains the following line:

```
fuelpump=200,201,300-310
```

The second file is called c:/si/userjar/usersession_fuelpump.properties. This file contains only the following line:

```
trace.buffer.size=5000
```

When you have created these two files, you must get them into a JAR file that is in the classpath. To create a new user.jar file that includes the usersessionrole.properties, and usersession_fuelpump.properties files, use the following command:

```
jar cvf user.jar *
```

To add the usersessionrole.properties, and usersession_fuelpump.properties files to an existing user.jar file, use the following command:

```
jar uvf user.jar usersessionrole.properties usersession_fuelpump.properties
```

The resulting user.jar file now must be copied to the c:\user directory on the 4690 controller (including setting file distribution attributes). In the case of TOF terminals, the TOF.BAT file must be rerun, or if bundling is being used, rebuild any terminal preload files that need to be rebuilt by running ADXPLDRB.386. Then reload the terminals.

Packaging your code

For user written code to be run, it must be included in one of the files in the classpath for the application. The default response files provided on the SI CD include several JAR files in the 4690 OS

Java classpath that are intended for user code. These JAR files include c:/user/user.jar for system wide properties, and c:/user/usersim.jar, for virtual sessions. By including your class and properties files in these JAR files, the response files do not have to be modified. The following example shows how to add some classes to the user.jar file.

Adding classes to a JAR file

For the purpose of building AEF user extensions in Java, all user Java class code should be in a directory hierarchy rooted at c:/si/userjar. For example, a user extension class called com.userco.retail.AEF.action.UserLogonActionImpl.java should be placed in c:/si/user/com/userco/retail/AEF/action/UserLogonActionImpl.java. When it is compiled, the **-d** option should be used to place the .class file in c:/si/userjar/com/userco/retail/AEF/action/UserLogonActionImpl.class.

To create a new user.jar file that includes all the user property files and class files in the userjar directory hierarchy, use the following command:

```
jar cvf user.jar *
```

Using the 4690 debugger for POS applications under CSS

This task describes the process for debugging a 4690 POS application such as SA or GSA while running under CSS. The steps required are as follows:

- Ensure the proper level of the Toshiba 4690 Application Debugger is installed on the 4690 controller. The debugger must be Version 4 Level 18 or later.
- Perform the necessary AEF configuration.
- Run the debugger. See "Running the debugger" on page 179.

Installing the 4690 debugger

The 4690 Application Debugger must be installed on the 4690 controller. The Debugger must be Version 4 Level 18 or later. See the Debugger installation diskette for installation instructions.

Configuring the AEF

There are two configuration properties that must be set to debug a virtual session. In the config.properties bundle, the debug.css.pos.appl must be set to true. In the session.properties bundle, the pos.sales.application property must be set to run EWTTERM.286.

1. Create (or edit) the file as follows:
   ```
   c:/si/userjar/userconfig.properties
   ```
2. Add the following line:
   ```
   debug.css.pos.appl=true
   ```
3. Choose the virtual session number to be debugged. Assuming the session number is 201, create or edit the following file:
   ```
   c:/si/userjar/usersession_201.properties
   ```
4. Add the following line:
   ```
   pos.sales.application=c:/debug/ewtterm.286
   ```

When you have created or edited the properties files, put them into a JAR file that is on the classpath for CSS. To create a new user.jar file that includes these files, use the following command:

```
jar cvf user.jar *
```

To add the files to an existing user.jar file, use the following command:

```
jar uvf user.jar usersession_201.properties userconfig.properties
```

On the 4690, stop CSS if it is running, and copy the user.jar file to the c:\user directory on the 4690 controller (including setting file distribution attributes), then restart CSS.

Running the debugger

To run the debugger on the 4690 controller, perform the following:

```
debug /Vnnn xxxxxxxx.286
```

where *nnn* is the session number to be debugged (201 in this example), and *xxxxxxxx* is the terminal sales application name.

Chapter 4. AEF enabling a POS application

AEF enabling a POS application

Follow these steps to enable a POS application to run in the SI environment:

1. Modify AEF configuration settings. It may be necessary to provide AEF configuration information, for example, default operator ID and passwords for terminal sessions.
2. Install POS application hooks. The POS application must include hooks to provide application XML data as appropriate. Events are expected for start and end of transactions, items added to the transaction, transaction totals updates, and so on.
3. Enable user extensions and additional features. The AEF must be programmed to perform POSAutomationProvider operations such as *logon*, *startTransaction*, *addItem*, *addTender* and so on. This involves providing action classes which understand the POS application. Each action class determines if the POS application is in the proper state to perform the action, sends the appropriate key sequences to the application, and defines the expected results. If the POS application includes user extensions or additional features, then the action classes must be extended or replaced so that the action is performed properly in the context of the user extensions.

AEF configuration settings

The AEF is configured via property name and value pairs that reside in various property files. See Appendix D, "AEF property files," on page 203 for information describing each of the configuration properties.

Some of these configuration properties give the AEF information about how to interact with the underlying POS application. The following sections describe these properties.

Default operator ID/passwords

One function of the POSAutomationProvider method is to allow a client program to logon to a terminal session. The client may specify an operator ID and password. If no operator ID and password is specified, the AEF uses the default operator ID and password which is configured for that session's terminal number. For details on defining default operator ID and passwords, see "Changing the default user ID and password for all terminals" on page 176.

Automatic manager override number

When a POSAutomationProvider method is called to perform some POS activity, the underlying POS application may generate an error or condition that can be overridden by using a manager's override. The POSAutomationProvider is configured by default to automatically perform the manager's override. See "Automation Bundle - error handling" on page 199 for information about the configuration property. When the POS application requires a valid manager override number, the AEF must be configured with that number. See "Automation Bundle - error handling" on page 199 for information about configuring a default manager override number.

Tender mappings

The AEF must be able to make correlations between tender descriptions (a string), and tender type and tender varieties (numeric values). For example, the Toshiba 4680-4690 Supermarket Application may have a tender definition for Credit with tender type equal to 4, and tender variety equal to 1.

For each tender definition in the application, a tender mapping needs to be configured in the AEF. See "Tender map bundle" on page 205 for information about configuring application tender mappings.

POS application hooks

The AEF receives information from the POS application in the form of XML events. Events are sent when an operator logs on or off, when a transaction starts or ends, when items are added or removed from the transaction, and so on.

See the POS applications section of the *Store Integrator User's Guide* (SC30-4085), for a complete list of Toshiba POS applications and features that are AEF-enabled.

For GSA and SA, the required terminal sales event hooks are included, but must be linked into the terminal sales program. Instructions for linking the AEF event hooks are included under *Selecting terminal sales components* in the *Store Integrator User's Guide* (SC30-4085)

For ACE, the required terminal sales event hooks are included in a special version of the terminal sales executable called JSIFTS10.386, which is included with the ACE package.

POS application events

The POS applications generate the following events which are transmitted to the AEF JVM. These XML events are used to generate Java events (see Event and Listener Types) that are triggered by the POSDataProvider. See Event and listener types for a list of POS application XML events.

If any GSA or SA user extensions require modification of terminal sales user exit or base source modules, these modules must be recompiled before the terminal sales link edit is performed. For more information refer to "Requirements for user exits, extensions, and source modifications" on page 187. It includes descriptions of the AEF user exits and provides other information regarding the preparation of the POS application programs for operation in an AEF environment.

Figure 76 on page 183 shows a typical AEF-enabled Toshiba POS application.

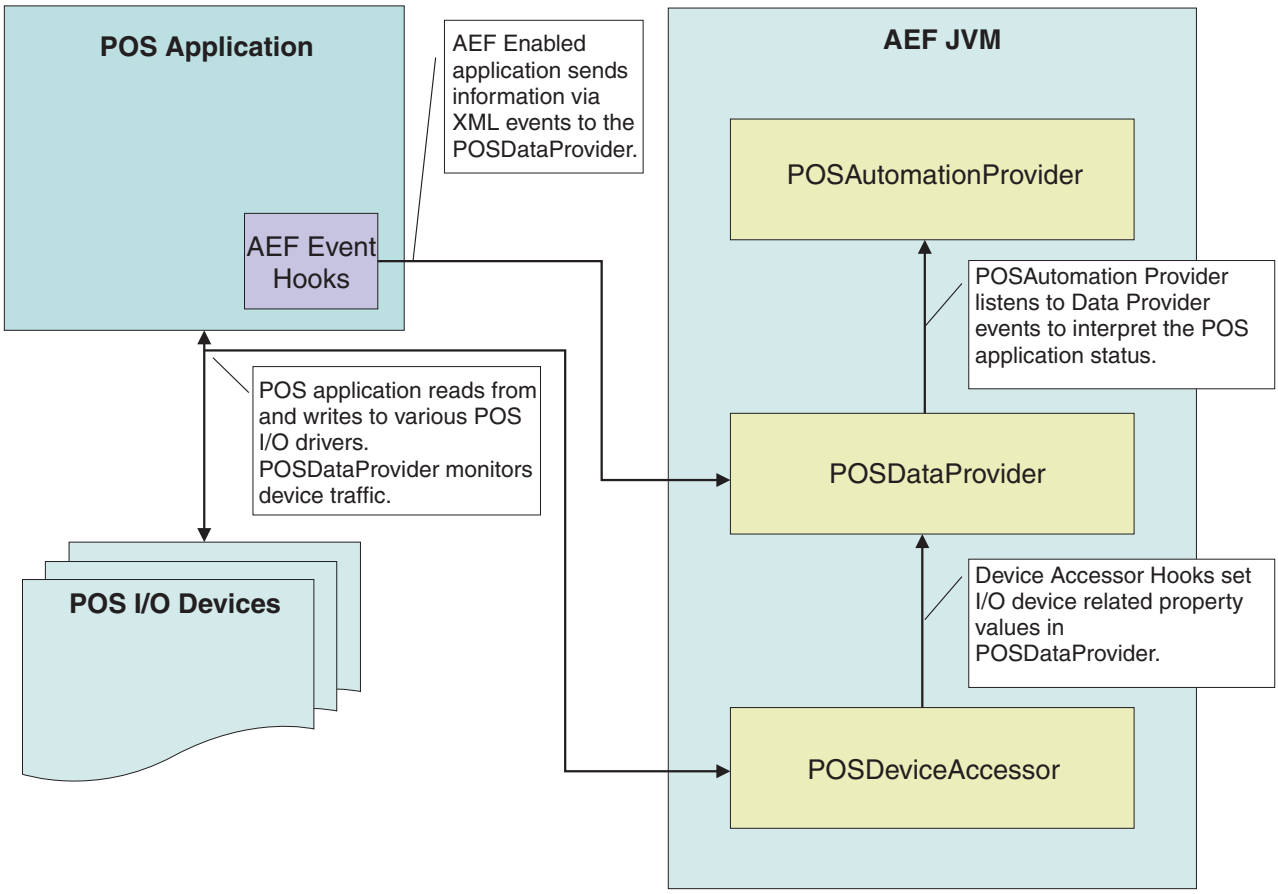
POS Application
AEF Enabled application sends information via XML events to the POSDataProvider.
AEF JVM
POSAutomationProvider
AEF Event Hooks
POSAutomation Provider listens to Data Provider events to interpret the POS application status.
POS application reads from and writes to various POS I/O drivers. POSDataProvider monitors device traffic.
POSDataProvider
POS I/O Devices
Device Accessor Hooks set I/O device related property values in POSDataProvider.
POSDeviceAccessor

Figure 76. AEF-enabled application

The AEF has two sources of information for POS application data:

1. The POS application is enhanced with AEF event hooks that transmit XML events to the AEF JVM. For example, events are transmitted when transactions are initiated or terminated, when items are added or removed from a transaction, when transaction totals are updated, when receipt print lines are generated, when the scale weight changes, when the application substate changes, and so on.
2. When POS I/O devices are redirected, the AEF may install Device Accessor hooks to monitor device data and status. For example, the AEF monitors the contents to the line displays, the 4690 Input/Output Processor (IOP) state, POSPrinter status, the last labels scanned, cash drawer status, the track data for the last card swiped, and so on.

Supermarket Application payment system hooks

Store Integrator requires that the 4690 Supermarket Application raise events that include data set by the payment system.

If your payment system is StorePay from Toshiba's National Retail Services Center (NRSC) you will need to obtain an updated version of StorePay that properly handles these events.

Otherwise, the following requirements must be satisfied by the payment system and or user exit code.

Table 33. Payment system requirements 1 of 7

Requirement	The POS application must raise the following transaction update event when it is waiting on a response from the host payment `<transactionUpdate paymentSentForHostProcessing="true"/>`

Table 33. Payment system requirements 1 of 7 (continued)

Notes	

Example code	`%INCLUDE JAVAGUIV.J86`
	`%INCLUDE JAVAGUIC.J86`
	
	`JAVA.PAYMENT.SENT.TO.HOST = -1`
	`CALL javaEvent(transactionUpdateMsg)`
	`JAVA.PAYMENT.SENT.TO.HOST = 0`

Table 34. Payment system requirements 2 of 7

Requirement	The POS application must raise a new substate (5008) when it's waiting on customer input at the pinpad. `<dataEvent dataCategory="POS_DEVICE" substate="5008"/>`
Notes	This substate should only be raised when terminal sales is waiting on the pinpad. For IBMEFT compatible pinpads, this is after the amount message (13.) has been sent and is waiting for authorization request (50.).

Example code	`%INCLUDE JAVAGUIV.J86`
	`%INCLUDE JAVAGUIC.J86`
	
	`jGuiSubState = waitingForPinpad`
	`CALL javaEvent(terminalSubStateMsg)`

Table 35. Payment system requirements 3 of 7

Requirement	The POS application must raise a new substate (5009) when it's waiting on customer signature at the pinpad. `<dataEvent dataCategory="POS_DEVICE" substate="5009"/>`
Notes	Since this event could occur during a display event which might have set a different jGuiSubState, it is recommended that the original jGuiSubState be saved so it can be restored after this event is fired (as shown in the example code below).

Example code	`%INCLUDE JAVAGUIV.J86`
	`%INCLUDE JAVAGUIC.J86`
	`integer*4  saveSubState`
	
	`saveSubState = jGuiSubState`
	`jGuiSubState = waitingForSignature`
	`CALL javaEvent(terminalSubStateMsg)`
	`jGuiSubState = saveSubState`

Table 36. Payment system requirements 4 of 7

Requirement	The POS application must raise a new substate (5010) instead of substate (11057) during the “ENTER ACCOUNT NUMBER” prompt if a pinpad is active.
Notes	The payment system code should set the global variable JAVA.PINPAD.ACTIVE to 0 if the pinpad is not present or goes offline. If the pinpad is present and online, or comes online, the payment system should set JAVA.PINPAD.ACTIVE to -1.
Example code	`%INCLUDE JAVAGUIV.J86` `JAVA.PINPAD.ACTIVE = -1`

Table 37. Payment system requirements 5 of 7

Requirement	The payment system must set global variables related to a host tender when a host tender is accepted or declined.
Notes	The variables in Table 38 on page 185 must be set.

Table 37. Payment system requirements 5 of 7 (continued)

Example code	`%INCLUDE JAVAGUIV.J86`
	`! Any of the JAVA.TENDER.XXXX variables may be`
	`! set by the payment system. Note that they`
	`! are cleared after the tender event is`
	`! raised.`

Table 38. Variables for payment system requirements 5 of 7

Variable	Datatype	Notes
JAVA.TENDER.CURRENCY$	String	See "Tender event" on page 260
JAVA.TENDER.REFNUM$		
JAVA.TENDER.CARDID$		
JAVA.TENDER.RESPONSE.CODE$		
JAVA.TENDER.MASKED.ACCOUNT.NUMBER$		
JAVA.TENDER.APPROVAL.CODE$		
JAVA.TENDER.RESPONSE.CODE.DESC$		
JAVA.TENDER.SEQUENCE.NUMBER$		
JAVA.TENDER.LEGAL.TEXT$		
JAVA.TENDER.REGE.PRINTED	Integer * 1	
JAVA.TENDER.SIGNATURE.NEEDED		

Table 39. Payment system requirements 6 of 7

Requirement	The POS application must raise a tender declined event if the host payment system declines a tender.
Notes	See "Tender declined" on page 261 for the format of the **tenderDeclined** event. The JAVA.TENDER.XXX global variables as defined in the prior section should be set before raising this event.

Example code	`%INCLUDE JAVAGUIV.J86`
	
	`! All the "JAVA.TENDER.XXX" global`
	`! variables should be set appropriately.`
	`! Note that all the "JAVA.TENDER.xxx"`
	`! global variables will be reset inside`
	`! the javaEvent routine.`
	`CALL javaEvent(tenderDeclinedMsg)`

Table 40. Payment system requirements 7 of 7

Requirement	Print line elements of cash receipt events must contain a **signatureLine** element. This element must be set to `true` when the print line is the tender franking second copy's signature line.
Notes	When the POS application generates a **printLine** event representing a signature line on the receipt, the global variable JAVA.SIGNATURE.LINE must be set to -1 by the payment system.

Example code	`%INCLUDE JAVAGUIV.J86`
	
	`JAVA.SIGNATURE.LINE = -1`
	`! Code which generates a java event`
	`! containing a printLine`
	`JAVA.SIGNATURE.LINE = 0`

Enabling user extensions and additional features

The process of AEF Enabling user extensions and additional application features involves two main aspects:

1. Modifying the POS application to send any additional required data to the AEF through existing or new XML messages.
2. Override or extend the AEF Automation Provider API as required to be able to deal with the POS application modifications.

Sending additional POS data to the AEF

An analysis of the AEF enablement requirements determines the additional POS application data that must be provided to the AEF through XML messages. It may be possible to extend existing XML messages as described in "Sending additional data to an event" on page 114. If this is not possible, then new XML messages may be sent to the AEF from the POS application as described in "Sending a new XML event" on page 115.

Overriding or extending the Automation Provider API

As previously explained in this guide, the Automation Provider API works by injecting input to the POS application, and monitoring properties such as I/O state, application substate, operator display contents, and so on to watch for conditions indicating success, failure, or errors which must be handled. Action classes are defined which check preconditions, inject key sequences, and wait for the results, or intermediate conditions which must be handled.

Each POS application must be examined in regards to the AEF to determine if the following might impact the AEF.

1. New operator prompts.
2. Key sequence changes.

Either of these might be the result of user extensions to the POS application, or applications features which are not AEF enabled.

New operator prompt:
These must be analyzed on a case by case basis. If the new prompt has the potential to occur during the executing of an Automation Provider API call, it probably has to be dealt with. Handling the new prompt depends on the nature of the prompt.

Typically, the action class must be overridden to deal with the prompt. In some cases, it may be possible to add an error helper configuration for the prompt without overriding the action class.

The process of overriding an action typically involves modifying a sequence's definition of expected good and bad conditions. The condition indicating the new prompt might be considered a good condition if it will be handled within the action. Alternatively, the condition indicating the new prompt might be defined as a bad condition if it will be handled through an error helper.

See "Extending existing Automation Provider APIs" on page 132 for an example of overriding an action class.

Key sequence changes:
If a user modification to the POS application changed a key sequence, it is necessary to reflect that change in the AEF. In the simplest case, the key sequence definition in the AEF can be modified in the sequence.properties bundle.

If the key sequence is a new key sequence, the same considerations for replacing the action implementation apply as discussed in the previous section.

If the key sequence has been modified to accept additional data, it might be necessary to modify the key sequence definition in the sequence.properties bundle, as well as replace the action implementation to send the additional data required by the modified key sequence. “Extending existing Automation Provider APIs” on page 132 is an example of overriding an action to handle a modified key sequence that requires additional data.

Requirements for user exits, extensions, and source modifications

To incorporate the necessary interfaces with the Java Virtual Machine into the terminal sales program under GSA or SA, the terminal sales application must be re-linked. Refer to the *Store Integrator User's Guide* (SC30-4085). If any user extensions require modification of terminal sales user exit or base source modules, these modules must first be recompiled.

Chapter 5. Programming with the POSProvider/POSServer API

This section describes how to set up the development environment and use the sample file. The POSProvider/POSServer infrastructure is also described.

Getting started

This section describes the POSProvider/POSServer API programming tasks.

Setting up the development environment

To use the POSProvider/POSServer API, you must first complete the AEF development environment install. To install the AEF, see Chapter 3, "Programming with the AEF," on page 93.

Additional files

The following files must be copied to your development system, in a directory called d:\si\si_jars:

- files from the 4690\AEF directory on the Store Integrator CD
- posprov.jar

User hierarchy

The directory structure for the POSProvider/POSServer is the same as the directory structure for the AEF. See Chapter 3, "Programming with the AEF," on page 93 for more information.

Using the sample file

Included with this document are precompiled files that demonstrate the features of the POSProvider/POSServer API. In this task, you will run the SampleSales client. This client makes a POSProvider, connects to the POSServer, gets the base item prices, and processes a transaction paying with a Visa credit card.

Running the SampleSales client

To run the SampleSales client, perform the following command:

```
java -classpath d:\si\user;d:\si\si_jars\posprov.jar;
d:\si\si_jars\aef.jar;d:\si\si_jars\siutil.jar;d:\si\si_jars\c_logging.jar;
-DaefIP=x.x.x.x -DaefTerm=tn sample.SampleSales
```

where *d* is the drive letter from which the zip containing this document was extracted, where *x.x.x.x* is the IP address of the host machine, and *tn* is the terminal number on the host machine.

Compiling the code

A precompiled version is provided. To modify the source code, you can compile it by browsing to the user directory, d:\si\user, and issuing the following command:

```
javac -classpath
d:\si\user;d:\si\si_jars\posprov.jar;d:\si\si_jars\aef.jar;
d:\si\si_jars\siutil.jar;d:\si\si_jars\c_logging.jar
sample/SampleSales.java
```

POSProvider/POSServer infrastructure

To make a connection to the point-of-sale system using the POSProvider/POSServer API, you must create the appropriate POSProvider object.

Creating a ProviderRequest

The first step required to creating a POSProvider is to make a ProviderRequest object. The ProviderRequest contains various bits of information needed to determine the kind of POSServer the POSProvider should to connect to. The ProviderRequest can be created using the RequestFactory factory. For this example we will use the KioskRequestFactory. The request can be made in the following way:

```
RequestFactory requestFactory = KioskRequestFactory.getInstance();
ProviderRequest providerRequest = requestFactory.makeProviderRequest();
```

The provider request accepts many different pieces of information which it uses when creating a POSProvider. The client is required to set two pieces of information; the ip address of the system to use, and the terminal number to use at that system. These can be set using the following methods:

```
setPOSTerminalNumber(String terminalNumber) - sets the terminal number
setIpAddress(String ipAddr) - sets the ip address
```

The provider request accepts the following optional data:

- backup ip address - setBackupIpAddress(String ipAddr)
- backup terminal - setBackupPOSTerminalNumber(String terminalNumber)
- client identifier - setClientIdentifier(String clientIdentifier)
- extended attribute - setExtendedAttribute(String name, String value)
- failover timeout - setFailoverTimeout(String timeout)
- payment confirm timeout - setPaymentConfirmTimeout(int seconds)
- pos timeout - setPOSTimeout(int timeout)
- request identifier - setRequestIdentifier(String requestIdentifier)
- request timeout - setRequestTimeout(int timeout)
- tender timeout - setTenderTimeout(int seconds)
- ip port - setIpPort(String port)

Note: Setting IP addresses, terminal number, or port number in this way overrides options set through properties files or the command line.

The POSProvider has the ability to connect to a backup point-of-sale system if the primary point-of-sale system is unable to handle incoming requests. To enable this functionality, the IP address of the backup point-of-sale system must be specified by using one of the following:

- '-DaefAltIP' option in the Java execution command
- 'posprov.alt.ip.addr' property in a posprov.properties overlay file
- setBackupIpAddress(String ipAddr) method on the ProviderRequest.

The backup must also have a terminal number specified. This is done by using the setBackupPOSTerminalNumber(String terminalNumber) method on the ProviderRequest.

Creating a POSProvider

After the ProviderRequest has been set up, it must be used to create the POSProvider object. This is done by obtaining an instance of the POSProviderFactory, and then using it and the ProviderRequest to create a POSProvider. The following is an example:

```
ProviderFactory providerFactory = KioskPOSProviderFactory.getInstance();
POSProvider provider = null;
try
{
   provider = providerFactory.makePOSProvider(providerRequest);
}
```

```
catch (Exception e)
{
   System.err.println("Unable to create instance of POSProvider");
}
```

Various exceptions can be thrown throughout the creation of the POSProvider. The list of exceptions can be found in Table 15 on page 60. Based on the exception received, the action to be taken is listed in Table 16 on page 60.

Testing the POSProvider connection

While the POSProvider is in use, it is important to make sure that the connection to the provider is still active. This is done by invoking the isPOSActive() command on the POSProvider. This test can prevent unnecessary attempts to send data to the server if it is not available. Also, if the POSProvider has enabled failover, this connectivity check will initiate it if the primary is down. The connection test can be performed in the following way:

```
Boolean isPOSActive = false;
try
{
    isPOSActive = provider.isPOSActive();
}
catch(Exception e)
{
   System.err.println("Unable to test provider connection");
}
```

Performing a sale

To perform a sales transaction, the client must first create a SalesRequest using the RequestFactory. The SalesRequest is used to bundle up a list of items, coupons, and returns to be processed by the point-of-sale application.

```
// Create an empty sales request
 SalesRequest salesRequest = requestFactory.makeSalesRequest();
```

Items to process are added after the SalesRequest is created. The RequestFactory is capable of making many different kinds of LineItems. Each type can be added to the SalesRequest in any order. A few examples are listed here. Use the JavaDoc to find more types of LineItems to create.

```
// Add the item identifiers
// A default item identifier is treated as an item barcode by the POS
salesRequest.addRequestLineItem(requestFactory.makeDefaultLineItem("1", "5"));
salesRequest.addRequestLineItem(requestFactory.makeDefaultLineItem("12", "2"));
salesRequest.addRequestLineItem(requestFactory.makeReturnLineItem("2", "1"));
```

A SalesTransaction object is required to submit the request to the point-of-sale application. To create a SalesTransaction, a TransactionRequest must first be created. This is done using the RequestFactory. Once the TransactionRequest has been created, it is then used in conjunction with the POSProvider to create the SalesTransaction. The SalesTransaction is then used to perform the sale using the SalesRequest.

```
// Create a transaction, a sales request and send it to the POS system.

SalesTransaction transaction = provider.makeSalesTransaction( requestFactory.makeTransactionRequest());
SalesResult salesResult = transaction.performSales(salesRequest);
```

Performing a payment

Performing a payment on the items that have been sold is required for any purchase that has a non-zero balance. This requirement applies to tender types that can be used for refunds, as well. The client can determine if a tender is required by checking the SalesResult object that was returned after performing the sale. This is achieved by using the isTenderRequired() method on the SalesResult.

If a tender is required, the client must first create a PaymentRequest object. This object is used to send the payment information to the POSServer for one tender. If using multiple tenders, the client must create a PaymentRequest for each tender. The PaymentRequest object is created through the RequestFactory.

Next, the client must create a TenderIdentifier object and add it to the PaymentRequest. This is done by using the setTenderIdentifier(TenderIdentifier) method on the PaymentRequest object. The client can use the RequestFactory to create a TenderIdentifier. Each type of TenderIdentifier requires that various fields are filled out, based on its interface. All tender type identifiers require the tender description to be filled in due to the variety of types of each tender. Credit, for example, can have a description of *visa* or *mastercard*. These accepted descriptions are in the AEF's tendermap.properties file.

The payment is performed through the SalesTransaction object's payBalance(PaymentRequest) method. When submitting payment using partial tenders, the payBalance() method must be invoked separately for each tender. Following is an example of the process used to perform a payment.

```
if(salesResult.isTenderRequired())
 {
  // Create an empty payment request and an empty credit payment info
  PaymentRequest paymentRequest = requestFactory.makePaymentRequest();
  CreditIdentifier creditCardInfo = requestFactory.makeCreditPaymentIdentifier();

  // Add the payment details to the credit card info
  String accountNumber = "4445222299990007";
  String date = "1004";
  creditCardInfo.setTenderDescription("visa");
  creditCardInfo.setAccountNumber(accountNumber);
  creditCardInfo.setExpirationDate(date);
  paymentRequest.setTenderIdentifier(creditCardInfo);

  // Execute the request on the point-of-sale system
  // Call is synchronous and client blocks until result is received
  PaymentResult paymentResult = transaction.payBalance(paymentRequest);
 }
```

Completing the transaction

When all items have been sold and all balances have been paid, the transaction must be confirmed. This insures that both the server and client have received all requests and results from each of the actions performed on the point-of-sale system. To confirm completion of the transaction, use the SalesTransaction object's confirm() method.

```
transaction.confirm();
```

Releasing POSProvider

When the client application has finished using the POSProvider, the client application must release it before terminating. By releasing the POSProvider, it cleans up the POSServer and gets it ready for the next client to establish a connection to it. The POSProvider is released using the release() method.

```
posProvider.release()
```

Force Release

If the POSProvider or Server gets into a state the client does not know how to get out of, a forced release can be issued to clean up the session and return it to a workable state. This is accomplished by invoking the forceRelease() method on the POSProvider.

```
posProvider.forceRelease()
```

A forceRelease() can be issued at any time during the transaction. It can also be issued in place of release() to ensure that the POSServer is cleaned up before the client terminates.

Appendix A. SA user exit code

The user exit module JAVAGUIU.BAS is installed into ADX_UPGM on the 4690 controller when you install the SA User Exit package. The relevant lines from this module are shown below:

```
%INCLUDE JGUIUVA.J86                          ! User variables
.
.
.

!————————————————————————————-
! subroutine javaAEFUserExit
!
! Parms: eventNum  - number of the event (application or user)
!        eventData - data associated with the event
!
!————————————————————————————-
SUB javaAEFUserExit(eventNum, eventData<) PUBLIC RECURSIVE ! javaGuiUserExit
  INTEGER*2 eventnum                          ! event number
  STRING    eventData<                        ! return data
  %INCLUDE JGUIU05.J86                        ! allow user to process
END SUB                                       ! javaGuiUserExit
```

In the previous example, the user exit subroutine is named javaAEFUserExit. This exit is called just before sending the XML message to the AEF. The eventNum indicates the cause of the event, and the eventData$ variable contains the XML message itself.

The following table lists the values of the eventNum variable:

Table 41. SA user exit event numbers

eventNum	**Value**
updateReceiptMsg	1
updateTotalMsg	2
updateScaleMsg	3
addAccountTenderMsg	4
addDiscountEntryMsg	5
addItemEntryMsg	6
addTenderEntryMsg	7
changeGivenMsg	8
endAccountingTransactionMsg	9
endSalesTransactionMsg	10
endTransactionMsg	11
operatorSignedOffMsg	12
operatorSignedOnMsg	13
optionsLoadedMsg	14
priceVerifyMsg	15
procedureCompletedMsg	16
procedureStartMsg	17
startAccountingTransactionMsg	18

Table 41. SA user exit event numbers (continued)

eventNum	Value
startSalesTransactionMsg	19
startTransactionMsg	20
reportOutputMsg	21
scrollMsg	22
displayVelocityCodesMsg	23
hideVelocityCodesMsg	24
redrawReceiptWindowMsg	25
updateStatusMsg	26
customerCardScannedMsg	27
javaKeysMsg	28
add2x20Msg	29
terminalSubStateMsg	30
optionsLoadingStartedMsg	31
optionsLoadingEndedMsg	32
optionsDataMsg	33
itemJustAddedMsg	34
voidTransactionMsg	35
foodStampTotalMsg	36
foreignTotalMsg	37
terminalConnectionChangeMsg	38
otrMsg	39
discountJustAddedMsg	40
operatorSpecialSignedOnMsg	41
updateTotalsDataMsg	42
aefEMreduemptionCpnMsg	43
aefUnitOfWorkMsg	44
operatorSpecialSignedOffMsg	45
procedureStartedMsg2	46
operatorDisplay	47
customerDisplay	48
noSaleStartMsg	49
noSaleEndMsg	50
jGuiEndAppEvents	51

Only a subset of the events in the table cause SA to generate an XML message for the AEF. Therefore, not all the event numbers listed above are passed through the javaAEFUserExit subroutine.

Appendix B. GSA user exit code

The SI installation CD contains the module EALAEFUE.BAS, which is installed into ADX_UPGM on the 4690 controller when you install the SI package.

```
!*************************************************************
function ?AEF.beforeXMLSend<(eventData<) public
  string    ?AEF.beforeXMLSend<
  string    eventData<
  ?AEF.beforeXMLSend< = eventData<
end function
```

As shown, the user exit subroutine is named ?AEF.beforeXMLSend$. This exit is called just before sending the XML message to the AEF. The eventData$ variable contains the XML message when may be modified by the user exit.

Appendix C. Error handling

Overview

Because of the design of the existing Toshiba POS applications, the same error message can be encountered while doing many different AEF automation operations. To provide AEF client applications with a consistent set of error messages, regardless of where the errors originated, the AEF automation APIs call a common error handler (AEFErrorHandler.java) when an error is encountered. Based on the severity of the error and the error handling mode of the application, the Error Handler will either try to clear the error, wait for the operator to clear the error, call a method on the client application (a registered callback), or throw an exception.

The error handler uses error helper classes. These error helpers can be generic helpers for a group or category of errors or specific for one error. Once the error helper has taken the appropriate action, it returns to the handler that determines if it is time to return to the calling application or continue to handle errors.

Besides the error helpers, the handler uses information from the AEF Error bundle, the Configuration bundle, the Class bundle, the Automation bundle, the Function Code bundle, Key Sequence bundle, the State bundle, the Sub-State bundle (GSA, SA and ACE), and the application's descriptors file (SA). How these bundles, files, and classes are used is discussed in the following sections.

Good and bad conditions

To understand how the error handler works, you must first understand the concept of good and bad conditions. When AEF tries to perform an automation command, a change in the application's state/substate is normally expected. An example of this is the Logon command. In SA, the application will be in State 2 if it is waiting for the user's ID during logon. When the ID is entered, the application goes to State 3 to wait for the password. If there is an error in the keying sequence or the ID is invalid then the application goes into State 1 (clear state). So in this case a good condition would be State 3 and a bad condition would be State 1.

In some cases the error handler knows that there was an error and that a good condition will never be achieved. In these cases, the error helper tries to get the application into a non-error condition. The non-error condition is derived from the bad conditions. This is done by converting them to their logical opposite. These opposite conditions are then concatenated into a logical "AND" statement.

Because the list of bad conditions contains the same state, sub-states and error messages for most of the automation commands, there is a standard list of bad conditions that is used. The BadConditions interface in the BadConditions.java file is used to define the behavior needed to implement the bad conditions list. A sample implementation of this interface is provided in the BadConditionsImpl class. This implementation is further extended for SA in the SABadConditionsImpl class.

AEFErrorHandler

The AEF Error Handler (AEFErrorHandler) is defined in AEFErrorHandler.java. The error handler is usually called by the AEF Automation commands when an error is encountered. It can also be called by the client application directly if the application determines that it is in an error condition. The following code fragment is typical of how the error handler is used within the AEF code.

Error handling

```
public int doSomething() throws AEFException
{
    int retVal = -1;
    ConditionLock lock = null;
    AEFAction keySequenceAction = null;
    Hashtable args = new Hashtable();
    AEFErrorHandler errorHandler = null;

    args.clear();
    args.put("SEQUENCE_ID", "doSometing");
    keySequenceAction = (AEFAction)(actionFactory.makeAction(new ActionRequest
        ("SimpleKeySequenceAction", args)));AbstractPropertyCondition[]
        goodConditions =
    {
        new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
            POSDeviceProperties.POS_SUB_STATE, trainingSubState)
    };
    lock = new ConditionLock();
    retVal = lock.performActionAndWait("wait-for-something",keySequenceAction,
        goodConditions,BadConditionsImpl.getInstance().getBadConditions(),getTimeout());
    if (retVal < 0)
    {
        errorHandler = new AEFErrorHandler("Doing Something");
        retVal = errorHandler.handleError(lock,keySequenceAction,goodConditions,
            BadConditionsImpl.getInstance().getBadConditions(),getTimeout(),retVal);
        if (retVal < 0)
        {
            throw errorHandler.getAllExceptions();
        }
    }
    return retVal;
}
```

In the example above, the performActionAndWait method of the ConditionLock class is called to do something. This method uses the third (goodConditions) and fourth (BadConditionsImpl.getInstance().getBadConditions()) parameters to determine if the action (the second parameter) completed successfully. If performActionAndWait returns zero or a positive number, then the action completed successfully and the error handler is not called. If performActionAndWait returns a negative number, then the action did not complete successfully and the handleError method of the error handler is called.

The handleError method takes as parameters the ConditionLock object used on the failed command, plus all the parameters passed to the failed performActionAndWait method. This gives the error handler all the information needed to handle the error and retry the command if need be. The handleError method returns zero or a positive number if the error can be cleared in such a way as to allow the original action to complete. It returns a negative number if the error cannot be cleared or is cleared in such a way that the original action cannot be completed successfully.

To communicate the error to the AEF client application, this code throws errorHandler.getAllExceptions(). The errorHandler.getAllExceptions method returns an AEFException object. The AEFException object may contain other AEFException objects within it. The getPreviousException method of the AEFException class can be used to get the AEFException inside an AEFException object. When this list of exceptions is built, the oldest exception is the innermost exception, while the outermost exception is the most recent one. In most cases, the outermost exception is a result of the inner exception, but this cannot be assumed; in some cases, multiple errors can be handled which are unrelated. One example is in tendering. Tendering with a large check may trigger an error because the amount exceeds a store limit. It may also trigger an error if the change due to the customer exceeds the store limit for change. These two exceptions are placed on the list as they were reported by the application and not because one caused the other.

Now that you have seen how it is used, here is what the basic flow is within the handleError method. The handleError method calls the handleSpecificError method while in a loop. Based on the return value of handleSpecificError, a wait is done to determine if the error helper has done all it can with this error. If the handleSpecificError returned a zero, then the helper cleared the error and the application should be in one of the good conditions passed in as a parameter. So the wait call will look for these good states. If the handleSpecificError returned -1, then the helper did not clear the error as expected. This means that the application's state is more than likely not in the list of good conditions that the caller passed in. Therefore, the wait call will look to make sure that the application is not in any of the bad states.

The condition to get out of the loop is satisfied once the wait returns a zero or a positive number. After the loop, a check is made to see if any of the error helpers added any exception to the exception list. If so, then the list of exceptions is thrown. Otherwise the return value from the last wait is returned.

The handleSpecificError method does several things. First, it checks to make sure that the maximum number of handled errors has not been reached. If so, then an exception is thrown. If the number of errors is still below the limit, then an AEFError object is created. This object gathers all the information needed to handle this error. With this information, the current error is compared to the last error to make sure that the error is not the same as the last error. If it is the same, then we are stuck in a loop and an exception is thrown. If it is a different error, then an ErrorHelper object is created and its handleError method is called. The return value of this call will be returned by the handleSpecificError method unless an exception is thrown.

Using the Configuration Bundle

The name of the first file in the Configuration Bundle is defined in the AEFConst.java file. The name of the constant is CONFIG_BUNDLE.

This is the entry in AEFConst.java:

public static String CONFIG_BUNDLE = "config";

Using this entry, the name of the first file in the configuration bundle is config.properties. Within this bundle, the error handler looks for theERROR_HANDLER_SLEEP_INTERVAL_2 property. The error helpers looks for the ERROR_HANDLER_SLEEP_INTERVAL_1 property. These properties are used to slow down AEF in cases where the hardware speeds may require it. The default value is 1 millisecond. Setting this value to zero causes the application to wait forever and should never be done under any circumstance.

Automation Bundle - error handling

The name of the first file in the Automation Bundle is defined in the AEFConst.java file. The name of the constant is AUTOMATION_PROPS_BUNDLE. Within this bundle, the error handler looks for the com\\ibm\\AEF\\automation\\error.max.errors.to.handle property.

This property is used to tell the error handler how many consecutive errors it should try to handle before throwing an exception. The default value for this property is 10 errors. The error helpers looks for the com\\ibm\\AEF\\automation\\error.handling.mode property. This property must be set to automatic, default, or callback.

This is the entry in AEFConst.java:

public static String AUTOMATION_PROPS_BUNDLE = "automation";

Using this entry, the name of the first file in the automation bundle is automation.properties.

The Class Bundle

The name of the first file in the Class Bundle is defined in the AEFConst.java file. The name of the constant is CLASSNAME_BUNDLE.

This is the entry in AEFConst.java:

public static String CLASSNAME_BUNDLE = "classes";

Using this entry, the name of the first file in the class bundle is classes.properties.

The error handler uses this bundle to get the class for the error helper needed to handle the current error. The error handler uses this bundle to get the class for the error helper needed to handle the current error. The error handler gets the key for this property from the Error Bundle.

Using the Error Bundle

The name of the first file in the Error Bundle is defined in the AEFConst.java file. The name of the constant is ERROR_BUNDLE.

This is the entry in AEFConst.java:

public static String ERROR_BUNDLE = "error";

Using this entry, the name of the first file in the error bundle is error.properties.

The error handler creates an AEFError object which in turn uses this bundle to get the characteristics of this error. These characteristics include the property name for the class key for the Error Helper (used with the Class Bundle), the Error Code and Extended Error codes (also defined in AEFConst.java) to be used when an exception needs to be thrown by the handler. There is also a field that includes additional error text that can be used to give the user more information than what is displayed on the 2x20 display. The following is an example of how the Supermarket's **B000** message is represented in the error bundle:

```
B000_ADDITIONAL_TEXT=An unrecoverable terminal error has been detected.
B000_HANDLING_CLASS_KEY=FatalErrorHelper
B000_ERROR_CODE=POS_APP_FAILURE
B000_EXTENDED_ERROR_CODE=NONE
```

Function Code Bundle

The name of the first file in the Function Code Bundle is defined in the AEFConst.java file. The name of the constant is FCODE_BUNDLE.

This is the entry in AEFConst.java:

public static String FCODE_BUNDLE = "fcode";

Using this entry, the name of the first file in the function code bundle is fcode.properties.

The error handler uses this bundle to get the function code number (application specific) for specific keys.

Key Sequence Bundle - error handling

The name of the first file in the Key Sequence Bundle is defined in the AEFConst.java file. The name of the constant is KEY_SEQUENCE_BUNDLE.

This is the entry in AEFConst.java:

public static String KEY_SEQUENCE_BUNDLE = "sequence";

Using this entry, the name of the first file in the key sequence bundle is sequence.properties.

The error handler uses this bundle to get the key sequence (application specific) needed to accomplish a task.

The State Bundle

The name of the first file in the State Bundle is defined in the AEFConst.java file. The name of the constant is STATE_BUNDLE.

This is the entry in AEFConst.java:

public static String STATE_BUNDLE = "states";

Using this entry, the name of the first file in the state bundle is states.properties.

The error handler uses this bundle to get the number associated with each application state (application specific). The state information is used to determine if we are in an error condition or if an automation command completed successfully.

Substate Bundle

The Substate Bundle is only used with applications that have sub-states. The first file of the Substate Bundle is defined in Substate.java. The entry is:

protected static String fileName = "appsubstates";

Using this, the first file in the substate bundle is appsubstates.properties. The error handler will use this bundle to get the number associated with each application substate (application specific). The substate information is used to determine if an error condition exists or if an automation command completed successfully.

Error bundle

The name of the first file in the Error Bundle is defined in the AEFConst.java file. The name of the constant is ERROR_BUNDLE.

This is the entry in AEFConst.java:

public static String ERROR_BUNDLE = "error";

Using this entry, the name of the first file in the error bundle is error.properties.

The error entries are used by AEF to determine how to handle each of the application errors. For a description of the process, see “AEF error handling” on page 34.

Error helpers

The job of the error helper is to try to resolve a specific error for the handler. In this manner, the Error Handler code does not need to be application or error specific.

The Error Helper methods are defined in the ErrorHelper interface (ErrorHelper.java). A sample implementation of this interface is provided in ErrorHelperImpl.java. The most important of the methods in this interface is the handleError method. This is the method that the Error Handler calls when it detects an error. This method returns a positive number if the error was cleared, or a negative number if the error could not be cleared. The handleError method does not throw exceptions; instead the exceptions are stored in an instance variable which the Error Handler gets upon return from the handleError call.

Some error helpers are very generic and can be used for many errors. Others are written specifically for one error. To allow the error handler to create the error helpers dynamically, the classes for all the error helpers must be defined in the Class bundle. The key in this bundle should match the xxxx_HANDLING_CLASS_KEY in the Error bundle. The following sections describe some of the error helpers.

Appendix D. AEF property files

Detailed information on configuring SI on the 4690 operating system is provided in the *Store Integrator User's Guide* (SC30-4085).

Editing property files

For any property entries that you add into your property files, the data after the "=" sign will be trimmed of all leading whitespace characters. However, the trailing whitespace characters *will not* be trimmed. It is important that you make sure that you do not have any trailing whitespace characters in your entries. Trailing spaces might not be detected by some editors. You can use the 4690 XE editor to eliminate these trailing whitespace characters.

Automation bundle

The Automation bundle specifies how automation calls should behave when errors are encountered or when an override is needed.

For each property, the default value is provided. A list of valid property values may also be specified by including the list inside [ ] characters. When a list is provided, the first value in the list becomes the default property value.

Class Bundle

The Class bundle tells AEF what Java class should be used to do a specific job.

For each property, a key is provided followed by an equal sign and a class name. These class names are used to create objects as needed. See "Editing property files." By changing the values for these keys, the classes of the objects created by AEF can be replaced with user classes.

Classes.properties

Classes.properties includes

- General Classes
- Listener support classes
- ApplicationDataConnector event classes (format: *xml-element = event class name*)
- Tender Line Item Implementation classes

appclasses.properties

The appclasses.properties for ACE include:

- General Classes
- Tender Actions
- Tender Line Item Implementation classes

The appclasses.properties for GSA and SA include:

- General Classes
- Tender Actions

Configuration Bundle

This bundle contains configuration properties for the AEF. These properties include settings for AEFBase and the SessionServer objects. The files that makeup the Configuration bundle are:

- config.properties
- appconfig.properties
- userconfig.properties

For each property, a key is provided followed by an equal sign and a value. If the value is %nodename%, then the value is evaluated to the controller ID or terminal number where the code is running. If the value is %platformSpecific%, then AEF processes the value based on what the platform is where the code is running.

See also "Editing property files" on page 203.

The Error bundle

The Error bundle specifies what error helpers should be used to handle specific errors.

Each error helper has four keys associated with each entry. These keys are ADDITIONAL_TEXT, HANDLING_CLASS_KEY, ERROR_CODE, and EXTENDED_ERROR_CODE. ADDITIONAL_TEXT provides the operator with additional information about the error. This information can describe how to clear the error or what information to collect to help in problem determination. The HANDLING_CLASS_KEY creates the error helper. The value of this key must be a key in the Class bundle. The ERROR_CODE, and EXTENDED_ERROR_CODE values are used as return values to the calling application to give an indication as to what errors were encountered. The value of these keys must be constants in AEFConst.java.

Using the Function code bundle

The Function code bundle maps a logical key name to a function code. For each property, a key is provided followed by an equal sign and a value. These values must match the function code values for the base application.

Internationalization bundle

The Internationalization bundle stores any text that is displayed to the end user of an AEF application. Therefore the text in the files that makeup this bundle must be translated into the local language.

For each property, a key is provided followed by an equal sign and a value. The double quotes around the values are used to tell the NLS team what text to change. They are removed by the I18nText.class when the text is read.

Key sequence bundle

The Key Sequence bundle specifies the key sequence required to do a particular POS operation on a terminal.

For each property, a key is provided followed by an equal sign and a value. If the value, or parts of the value, is surrounded by { }, then this text is replaced by a variable at runtime. If the value, or part of the value, is surrounded by < >, then the text is replaced by the function code specified in the function code bundle for that key. If the value, or part of the value, is surrounded by [$], then the text is replaced by the appropriate POSDataProvider property value at runtime. Any other text is assumed to be part of the keying sequence and is sent to the application without modification.

Logon bundle

The Logon bundle specifies a default user ID and password combination for specific terminals.

Note: It is recommended that you encrypt your logon properties file for security. See the *Store Integrator User's Guide* (SC30-4085). for additional information.

For each property, a key is provided followed by an equal sign, a number, a comma, and another number. The key is the terminal number. The first number after the equal sign is the operator ID to use. The number after the comma is the password to use.

Session bundle

The Session bundle configures session specific settings.

For each property, a key is provided followed by an equal sign and a value.

Session.properties information

The session.properties file contains session settings such as debug and trace properties, tuning properties, and others.

Note:

1. Extension class must use a default constructor for instantiation.
2. Extension class must implement the com.ibm.retail.AEF.session.SessionExtension interface.
3. Session extension objects are created after SessionReady event in AEFSession.
4. The AEFSession object is passed to the extension using the setSession method.

State bundle

The State bundle maps a logical application state to a state number.

For each property, a key is provided followed by an equal sign and a value. The values for these keys must match the valid state values for the base application.

Substate bundle - AEF

The Substate bundle maps a logical application substate to a substate number.

For each property, a key is provided followed by an equal sign and a value. These values must match the valid substate values for the base application.

Note: For ACE, additional keys have been added to match a substate to an error message.

Tender map bundle

The Tender Map bundle maps a tender name to an application specific tender type. This is done by concatenating the tenderType and tenderVariety fields of the tender XML event from the POS application. This concatenated key is then looked up in the classes.properties file to determine the Java classname to use to represent the tender line item.

Each entry in these files contains the following four fields.

Key This value is used as keys in the classes.properties chain. It generates the tender line item by appending Tender to the key to look up the tender line item class in classes.properties. It generates the add tender action class key by prepending Add and appending TenderAction, so

that AddKeyTenderAction is the resulting key for use in classes.properties. It also generates the void tender action class key by prepending Void and appending TenderAction, so that VoidKeyTenderAction is the resulting key for use in classes.properties.

Description
This should match the tender description as defined in the application. When the application provides its tender definitions, the tender type and variety listed in the property file may be overridden by the application data if the description matches (ignoring case and punctuation differences).

Note: This is the only field that should be translated.

Type The tender type, as defined by the POS application.

Variety
The tender variety, as defined by the POS application.

Appendix E. Data Provider properties

This appendix lists the property values maintained by the Data Provider. Each table corresponds to a category.

The properties interfaces form an inheritance structure. Therefore some properties defined in a base interface (such as the LineItemProperties interface) apply to any of the child interfaces.

Note: Not all applications support all the listed properties.

CouponProperties

The constant CouponProperties.CATEGORY can be used for the category name. These properties are valid for the last coupon in the transaction.

Property Name Constant	Property Name	Type	Notes
COUPON_TYPE	couponType	String	Store or Manufacturer coupon.
UNIT_PRICE	unitPrice	String	
DEAL_QUANTITY	dealQuantity	String	
WEIGHT	weight	String	
VALUE	value	String	The total value of the coupon. This depends on the pricing method, whether the coupon applies to a weighted item, and so on.
DEAL_PRICE	dealPrice	String	Applies to deal price pricing method only.
REDUCED_PRICE	reducedPrice	String	Applies to reduced price pricing method only.
MULTI_PRICING_GROUP	groupID	String	Applies to group pricing methods only.
AGE_RESTRICTION	requiredAge	String	The required age (in years) to redeem the coupon.
REDUCES_FOODSTAMP_BALANCE	reducesFoodstampBalanceDue	String	*true* if this coupon reduces the foodstamp balance due.
COUPON_REPEAT_ALLOWED	itemRepeatAllowed	String	*true* if the operator can repeat the coupon in the transaction by pressing <Enter>.
REDUCES_TAX_DUE	reducesTaxDue	String	*true* if the coupon reduces the tax balance due.

Data Provider properties

Property Name Constant	Property Name	Type	Notes
PRICING_METHOD	pricingMethod	String	Property value is the name of the coupon pricing method. Values may be: • unit • dealQuantity • basePlus1 • groupThreshold • reducedDeal • increasedDeal • alias
MANUFACTURER_NUMBER	manufacturerNumber	String	Coupon manufacturer number if coupon is a manufacturer coupon.
IS_VOIDED	isVoided	String	*true* if the coupon has been voided.
ITEM_TAXABLE	itemTaxable	String	*true* if the coupon is taxable. If the coupon is taxable, it does not reduce the tax balance due.
IS_DEPOSIT	isDeposit	String	*true* if the coupon represents a deposit item.
IS_REFUNDED	isRefunded	String	*true* if the coupon has been refunded.
ITEM_MODIFIER	itemModifier	String	Value represents application-unique information.
RESTRICTED_PERIODS	restrict	Collection	Value is a collection of TimeInterval elements representing the period of time during which the coupon either cannot be redeemed, or must be redeemed.
WIC_ELIGIBLE	wicEligible	String	*true* if the coupon is WIC eligible, *false* if not.
LINKED_ITEM_ID	linkedItemID	String	The identifier of the item that is linked from this coupon.
LINKED_ITEM_ID_QUALIFIER	linkedItemIDQualifier	String	Application specific information about the linked item ID.
DEPARTMENT_NUMBER	departmentNumber	String	The department that this coupon is applied to.
VOLUME	volume	String	The volume unit of measure that the coupon applies to.

CustomerProperties

The constant CustomerProperties.CATEGORY can be used for the category name. These properties are valid for the last customer loyalty number entered for the transaction.

Property Name Constant	Property Name	Type	Notes
CUSTOMER_ID	id	String	Customer's loyalty number.
ID_TYPE	idType	String	Indicates if the customer number was the "primary" loyalty number, or an "alternate" loyalty number.
ID_EXPIRATION_DATE	idExpirationDate	String	
CUSTOMER_NAME	name	String	Format is unspecified.
CUSTOMER_EMAIL	email	String	Not supported.
TARGETED_COUPONS	targetedCoupons	Collection	Collection of strings where each string in the collection is an item code of a targeted coupon.
POINTS_TOTALS	pointsTotals	Collection	Collection of PointsTotal objects. Each represents a loyalty points category.
POINTS_BALANCES	pointsBalances	Collection	Collection of string objects. Each element is the total number of points in a loyalty category.
MESSAGES	messages	Collection	Collection of string objects. Each element is a loyalty message targeted for the customer.
MESSAGE	message	String	Loyalty message that is displayed when the customer ID is added to the transaction.

DiscountProperties

The constant DiscountProperties.CATEGORY can be used for the category name. These properties are valid for the last line item or transaction discount entered for the transaction.

Property Name Constant	Property Name	Type	Notes
DISCOUNT_METHOD	discountMethod	String	Either *percent* or *allowance*. An allowance is a fixed amount discount.
DISCOUNT_AMOUNT	amount	String	The net amount of the discount.
DISCOUNT_RATE	rate	String	For a percent discount, this is the percentage. Format is a percentage, for example, 15% is represented as "15.0".
DISCOUNT_REDUCES_TAX	reducesTaxBalanceDue	String	*true* if the discount causes the tax due to be decreased. The tax due is decreased when the tax is calculated after the discount has been applied.
IS_TRANSACTION_DISCOUNT	isTransactionDiscount	String	*true* if the discount applies to the entire transaction.
IS_VOIDED	isVoided	String	*true* if the discount has been voided.
DISCOUNT_TYPE	discountType	String	Application specific information.

Property Name Constant	Property Name	Type	Notes
DISCOUNT_REASON	discountReason	String	Application specific discount reason code.

ItemSalesProperties

The constant ItemSalesProperties.CATEGORY can be used for the category name. These properties are valid for the last item entered for the transaction.

Property Name Constant	Property Name	Type	Notes
UNIT_PRICE	unitPrice	String	Applies if pricing method is *unit*.
WEIGHT	weight	String	The item weight if sold by weight.
EXTENDED_PRICE	extendedPrice	String	
ITEM_REPEAT_ALLOWED	itemRepeatAllowed	String	*true* if the operator can sell another of the same item using a repeat key sequence.
IS_RETURN	isReturn	String	*true* if the line item is an item return.
RETURN_REASON	returnReason	String	Application specific return reason code. Applies only if the line item is an item return.
REQUIRED_AGE	requiredAge	String	The customer minimum age (in years) to purchase this item. If no value, the item is not age restricted.
RESTRICTED_PERIODS	restrict	Collection	Value is a Collection of TimeInterval elements representing the period of time during which the item cannot be purchased, or must be purchased.
FOOD_STAMP_ELIGIBLE	foodstampEligible	String	*true* if the item may be purchased with foodstamps.
WIC_ELIGIBLE	wicEligible	String	*true* if the item is WIC eligible, *false* if not.
LINKED_ITEM_ID	linkedItemID	String	Item code of an item that is automatically sold when this item is sold.
LINKED_ITEM_ID_QUALIFIER	linkedItemIDQualifier	String	Application specific information about the linked item ID.
DEAL_QUANTITY	dealQuantity	String	The quantity involved in dealQuantity-pricing or other pricing methods that require a quantity.
DEAL_PRICE	dealPrice	String	The price for a deal pricing method. For example, if a deal is 3 for $1.00, the dealPrice is "1.00".
REDUCED_PRICE	reducedPrice	String	A threshold price used by certain pricing methods.

Property Name Constant	Property Name	Type	Notes
MULTI_PRICING_GROUP	groupID	String	Identifies the group to which that the item belongs for multigroup pricing schemes.
PRICING_METHOD	pricingMethod	String	The pricing method. For Toshiba 4690 Supermarket Application, the values may be: • unit • dealQuantity • basePlus1 • groupThreshold • reducedDeal • increasedDeal • alias
IS_VOIDED	isVoided	String	*true* if the line item is an item void.
ITEM_TAXABLE	itemTaxable	String	*true* if the item is taxable.
IS_DEPOSIT	isDeposit	String	*true* if the item is a deposit item.
IS_REFUNDED	isRefunded	String	*true* if the item is a line item refund.
ENTERED_PRICE_USED	enteredPriceUsed	String	*true* if the operator entered the item price (because the item was a price required item, or because of a price override).
ORIGINAL_SALESPERSON	originalSalesperson	String	If the application can provide this information, it is the salesperson who originally rang up the item. Usually used for item returns.
EAS_ITEM_TYPE	EASItem	String	*true* if this is an EAS item, *false* if not.
DEPARTMENT_NUMBER	deparmentNumber	String	The department that this item belongs to.
ITEM_TYPE	itemType	String	The type of the item, such as NormalSaleItem.
VOLUME	volume	String	The volume unit of measure of the item.
NET_UNIT_PRICE	netUnitPrice	String	The net unit price of the item, for fuel this takes into account any coupons
NET_EXTENDED_PRICE	netExtendedPrice	String	The net extended price, for fuel this is the total price of the sale at the pump.
AUTHORIZATION_CODE	authorizationCode	String	The authorization code associated with this item, for example, a car wash authorization code.

LineItemProperties

The properties in the LineItemProperties interface apply to all line items. The following properties apply to the last LineItem in the transaction.

Property Name Constant	Property Name	Type	Notes
ITEM_ID	itemID	String	The item identifier. May be UPC, velocity code, and so on.
ITEM_ID_QUALIFIER	itemIDQualifier	String	Indicates the format of the item code (may be scanned or keyed). For Toshiba 4690 Supermarket Application, the values may be: • ScannedItemCode • KeyedItemCode • ItemLookupKeyed • LinkedItemCode • WandedItemCode
QUANTITY	quantity	String	
DESCRIPTION	description	String	
IS_VOIDED	isVoided	String	*true* if the line item is voided.
IS_DEPOSIT	isDeposit	String	*true* if the line item is a deposit item.
IS_REFUNDED	isRefunded	String	*true* if the line item is refunded.
PRINT_LINES	printLines	Collection	Value is a Collection of string elements. Each represents a print line on the receipt. Any unprintable characters are filtered from the strings.
RAW_PRINT_LINES	rawPrintLines	Collection	Value is a collection of string elements. Each represents a print line on the receipt. Unprintable characters are transformed as [*nn*] where *nn* is the ASCII value of the character.

OperatorAuthorizationProperties

The constant OperatorAuthorizationProperties.CATEGORY can be used for the category name. These properties are valid for the operator who is logged onto the session.

Authorization properties (common)

The following table lists authorization properties common to two or more of the Toshiba POS applications (GSA, SA, and ACE).

Table 42. Data Provider authorization properties (common)

Property Name Constant	Property Name	Type	Options
ITEM_PRICE_CHANGE_ALLOWED	itemPriceChangeAllowed	string	true or false
LAYAWAY_ALLOWED	layawayAllowed	string	true or false
LOAN_ALLOWED	loanAllowed	string	true or false
OBTAIN_ITEM_MOVEMENT_ALLOWED	obtainItemMovementAllowed	string	true or false

Table 42. Data Provider authorization properties (common) (continued)

Property Name Constant	Property Name	Type	Options
OBTAIN_DEPT_TOTALS_ALLOWED	deptTotalsReportAllowed	string	true or false
OPERATOR_TRAINING_ALLOWED	operatorTrainingAllowed	string	true or false
REENTRY_OFFLINE_SALES_ALLOWED	reentryOfflineSalesAllowed	string	true or false
REGISTER_READOUT_ALLOWED	registerReadoutAllowed	string	true or false
RETURNS_ALLOWED	returnsAllowed	string	true or false
DISCOUNTS_ALLOWED	discountsAllowed	string	true or false
VOID_TRANSACTION_ALLOWED	voidTransactionAllowed	string	true or false
PRICE_VERIFICATION_ALLOWED	priceVerifyAllowed	string	true or false
_MONITOR_ALLOWED	MonitorAllowed	string	true or false
TENDER_LISTING_ALLOWED	tenderListingAllowed	string	true or false

Authorization properties (GSA)

The following table lists authorization properties related to the Toshiba POS application GSA.

Table 43. Data Provider authorization properties (GSA)

Property Name Constant	Property Name	Type	Options
PASSWORD_REQUIRED	passwordRequired	string	true or false
ALLOWANCE_ALLOWED	allowanceAllowed	string	true or false
CASH_TRANSACTION_ALLOWED	cashTransactionAllowed	string	true or false
CASH_DOC_ALLOWED	cashDocumentTransactionAllowed	string	true or false
CASH_SPEC_ALLOWED	cashSpecialTransactionAllowed	string	true or false
CHARGE_TYPE_6_ALLOWED	chargeType6TransactionAllowed	string	true or false
CHARGE_TYPE_7_ALLOWED	chargeType7TransactionAllowed	string	true or false
CHARGE_TYPE_8_ALLOWED	chargeType8TransactionAllowed	string	true or false
CHARGE_TYPE_9_ALLOWED	chargeType9TransactionAllowed	string	true or false
TENDER_REMOVAL_ALLOWED	tenderRemovalAllowed	string	true or false

Table 43. Data Provider authorization properties (GSA) (continued)

Property Name Constant	Property Name	Type	Options
COD_ALLOWED	codTransactionAllowed	string	true or false
DELAYED_PRICE_CHANGE_ALLOWED	delayedPriceChangeAllowed	string	true or false
REGISTER_RESET_ALLOWED	registerResetAllowed	string	true or false
SET_TRANSACTION_NUMBER_ALLOWED	setTransactionNumberAllowed	string	true or false
SUSPEND_TRANSACTION_ALLOWED	suspendTransactionAllowed	string	true or false
TPL_TRANSACTION_ALLOWED	tplTransactionAllowed	string	true or false
VOID_PREVIOUS_BY_LINE_ALLOWED	voidPreviousItemAllowed	string	true or false
VOID_PREVIOUS_TRANSACTION_ALLOWED	voidPreviousTransactionAllowed	string	true or false
QUERY_EXCHANGE_RATE_ALLOWED	queryExchangeRateAllowed	string	true or false
NO_SALE_ALLOWED	nosaleOpenCashDrawerAllowed	string	true or false
WITHDRAWALS_TRANSACTION_ALLOWED	pickupAllowed	string	true or false
CASH_COUNT_ALLOWED	tenderCountAllowed	string	true or false

Authorization properties (SA and ACE)

Table 44 lists authorization properties related to the Toshiba POS applications SA and ACE.

Table 44. Data Provider authorization properties (SA and ACE)

Property Name Constant	Property Name	Type	Options
CHECKOUT_TRANSACTION_ALLOWED	salesAllowed	string	true or false
TENDER_CASHING_ALLOWED	tenderCashingAllowed	string	true or false
TENDER_EXCHANGE_ALLOWED	tenderExchangeAllowed	string	true or false
PICKUP_TRANSACTION_ALLOWED	pickupAllowed	string	true or false
TERMINAL_TRANSFER_ALLOWED	TransferAllowed	string	true or false
TENDER_COUNT_ALLOWED	tenderCountAllowed	string	true or false
RETURN_ITEM_ALLOWED	returnItemAllowed	string	true or false
WIC_TRANSACTION_ALLOWED	wicTransactionAllowed	string	true or false

Table 44. Data Provider authorization properties (SA and ACE) (continued)

Property Name Constant	Property Name	Type	Options
USER_NON_SALES_1_ALLOWED	userNonsales1Allowed	string	true or false
USER_NON_SALES_2_ALLOWED	userNonsales2Allowed	string	true or false
NO_SALE_OPEN_CASH_DRAWER_ALLOWED	nosaleOpenCashDrawerAllowed	string	true or false
NO_SALE_TENDER_REMOVAL_ALLOWED	nosaleTenderRemovalAllowed	string	true or false
NO_SALE_TILL_EXCHANGE_ALLOWED	nosaleTillExchangeAllowed	string	true or false
NO_SALE_TENDER_VERIFY_ALLOWED	nosaleTenderVerifyAllowed	string	true or false
NO_SALE_TILL_REPORT_ALLOWED	nosaleTillReportAllowed	string	true or false
NO_SALE_PRICE_VERIFY_ALLOWED	nosalePriceVerifyAllowed	string	true or false
REFUNDS_ALLOWED	refundsAllowed	string	true or false
MISC_ITEM_PAYOUTS_ALLOWED	miscItemPayoutAllowed	string	true or false
IMMEDIATE_IR_CHANGES_ALLOWED	immediateItemChangeAllowed	string	true or false
DELAYED_IR_CHANGES_ALLOWED	delayedItemChangeAllowed	string	true or false
MORE_THAN_PRICE_CHANGES_ALLOWED	moreThanPriceChangesAllowed	string	true or false
MANAGERS_PROCEDURES_ALLOWED	managersProceduresAllowed	string	true or false
REPRINT_TENDER_RECEIPT_ALLOWED	reprintTenderReceiptAllowed	string	true or false
USER_FUNCTION_1_ALLOWED	userFunction1Allowed	string	true or false
FRONT_END_CASHIER_ALLOWED	frontEndCashierAllowed	string	true or false
DEPT_TOTALS_REPORT_ALLOWED	deptTotalsReportAllowed	string	true or false

OptionsProperties

The constant OptionsProperties.CATEGORY can be used for the category name. These properties reflect the latest personalization options loaded by the POS application.

Property Name Constant	Property Name	Type	Notes
STORE_DEFINITION	store	(S)	The StoreDefinition contains information such as store number, name, address, and phone numbers.

Data Provider properties

Property Name Constant	Property Name	Type	Notes
TENDER_DEFINITION	tenderDefinition	Collection	Value is a collection of TenderDefinition elements. Each element contains the POS application's definition of a tender.
PRICE_OVERRIDE_REASON	priceOverrideReason	Collection	Value is a collection of ReasonCode elements. This collection represents the price override reason codes configured in the POS application.
REFUND_REASON	refundReason	Collection	Value is a collection of ReasonCode elements. This collection represents the refund reason codes configured in the POS application.
VOID_REASON	voidReason	Collection	Value is a collection of ReasonCode elements. This collection represents the void reason codes configured in the POS application.
SECURITY_VOID_REASON	SecurityVoidReasonCode	String	The security void reason code to be used to indicate automatic voids.
TARE_CODE	tareCode	Collection	Value is a collection of TareCode elements. This collection represents the tare codes configured in the POS application.
VAT_TAX_CODE	vatTaxCode	Collection	Value is a collection of TaxCode elements. This collection represents the VAT tax codes configured in the POS application.
ALTERNATE_TAX_CODE	alternateTaxCode	Collection	Value is a collection of TaxCodeelements. This collection represents the tax codes configured in the POS application which represent alternate tax tables.
MANUAL_TAX_CODE	manualTaxCode	Collection	Value is a collection of TaxCode elements. This collection represents the tax codes configured in the POS application which represent manual tax amount entries.
NO_TAX_CODE	nolTaxCode	Collection	Value is a collection of TaxCode elements. This collection represents the tax codes configured in the POS application which represent zero tax entries.

Property Name Constant	Property Name	Type	Notes
TRANSACTION_DISCOUNT_ REASON_CODE	transactionDiscountReason	Collection	Value is a collection of TransactionDiscount ReasonCode elements. This collection represents the transaction discount reason codes configured in the POS application.
ITEM_DISCOUNT_REASON_CODE	itemDiscountReason	Collection	Value is a collection of DiscountReasonCode elements. This collection represents the item discount reason codes configured in the POS application.
DEPARTMENT_DEFINITION	departmentDefinition	Collection	Value is a collection of DepartmentDefinition elements. This collection represents the departments configured in the POS application.
TERMINAL_OPTIONS	Options	Collection	Value is a collection of Options elements. This collection represents options configured in the POS application.
STORE_OPTIONS	storeOptions	Collection	Value is a collection of StoreOptions elements. This collection represents store options configured in the POS application.
FUEL_OPTIONS	fuelOptions	Collection	Value is a collection of FuelOptions elements. This collection represents fuel options configured in the POS application.
DEPARTMENT	department	Collection	Value is a collection of Department objects which define each department's number and description.
LOGO_DEFINITION	logoDefinition	Collection	Value is a collection of LogoDefinition elements. Each element contains a mapping between a logo name and its preloaded index in the printer.

PointsProperties

The constant PointsProperties.CATEGORY can be used for the category name. These properties reflect the latest loyalty points that were rewarded or redeemed within the POS application transaction.

Property Name Constant	Property Name	Type	Notes
ID	id	String	Identifies the points. For Toshiba Global Commerce Solutions applications such as SA and ACE, this is the points item code.
ID_QUALIFIER	idQualifier	String	Indicates the format of the points identifier.
TYPE	type	String	Refers to the loyalty points category.
TOTAL	points	String	This is the number of points rewarded or redeemed.
POINTS_REDEEMED	redeemed	String	*true* if the points were redeemed in the transaction, *false* if they were awarded.
POINTS_VOIDED	isVoided	String	*true* if the points were voided in the transaction.
ADDITIONAL_POINTS_TOTALS	pointsTotals	Collection	Value is a Collection of PointsTotal elements. Each element represents the new points total for each points category.

POSDeviceProperties

The constant POSDeviceProperties.CATEGORY can be used for the category name. These properties reflect POS I/O device and I/O processor attributes.

Note: The appropriate devices must be configured as redirected for the property values to be set properly.

Property Name Constant	Property Name	Type	Notes
ANPROMPT_LINE1	ANPROMPT_LINE	String	Contains the contents of the first line of the operator line display.
ANPROMPT_LINE2	ANPROMPT_LINE2	String	Contains the contents of the second line of the operator line display.
OPERATOR_PROMPT_LINE1	ANPROMPT_LINE	String	Contains the contents of the first line of the operator line display.
OPERATOR_PROMPT_LINE1	ANPROMPT_LINE2	String	Contains the contents of the second line of the operator line display.
CUST_PROMPT_LINE1	CUST_PROMPT_LINE	String	Contains the contents of the first line of the customer line display.

Property Name Constant	Property Name	Type	Notes
CUST_PROMPT_LINE2	CUST_PROMPT_LINE2	String	Contains the contents of the second line of the operator line display.
OPERATOR_DISPLAY	OPERATOR_DISPLAY	String	Indicates which line display is being used by the POS application as the operator display. Possible values are: • LineDisplay1 • LineDisplay2 • LineDisplay3
CUSTOMER_DISPLAY	CUSTOMER_DISPLAY	String	Indicates which line display is being used by the POS application as the customer display. Possible values are: • LineDisplay1 • LineDisplay2 • LineDisplay3
POS_SUB_STATE	subState	String	A numeric value which indicates the current substate of the POS application.
POS_STATE	POS_STATE	String	A numeric value indicating the current I/O processor state of the POS application.
CHANGE_TO_CURRENT	CHANGE_TO_CURRENT	String	*true* indicates that the last POS_STATE change was a transition from a state to the same state. **Note:** This is a specific definition in the application state table. The function code must be defined to "CHANGE TO CURRENT". State changes from a state to the same state for other reasons do not cause this flag to be set.
SYSTEM_BUSY	SYSTEM_BUSY	String	*true* indicates that the I/O processor input queue is locked. No keyboard or scanner input is accepted.
PAPER_CUT	paperCut	String	*true* indicates that the POS application has sent a paper-cut command to the receipt printer.
PRINT_LINE	printLine	String	A string containing the last print line sent to the receipt printer.
PRINT_LINE_ARRAY	PRINT_LINE_ARRAY	ArrayList	Value is an ArrayList of String elements. Each String element is a cash receipt print line for the current transaction.

Data Provider properties

Property Name Constant	Property Name	Type	Notes
RAW_PRINT_LINE_ARRAY	RAW_PRINT_LINE_ARRAY	ArrayList	Value is an ArrayList of String elements. Each String element is a raw cash receipt print line for the current transaction. "Raw" means that print control characters are included with the text.
PRINT_LINE_FEEDS	lineFeeds	String	Value is numeric, indicating the last number of line feeds sent to the receipt printer.
APP_WILL_PROVIDE_DATA	APP_WILL_PROVIDE_DATA	String	*true* indicates that the POS application provides the cash receipt data through XML events. *false* indicates that the cash receipt data comes from a Device Accessor printer hook. The printer hook approach causes a delay in receiving print lines from applications using clean receipt or post printing. The value for this property is provided by the POS application.
SCALE_WEIGHT_VALUE	scaleWeightValue	String	Value is numeric, showing the weight of the item on the scale.
SCALE_WEIGHT_UNIT	scaleWeightUnit	String	Indicates the units of measure of the scale weight.
SCALE_WEIGHT_LABELS	scaleWeightLabels	String	Additional scale display text as required by Weights and Measures.
KEYLOCK_IS_SUPERVISOR	KEYLOCK_IS_SUPERVISOR	String	*true* indicates that the keylock is in the manager position. *false* indicates the keylock is in the normal position.
CASH_DRAWER_OPEN	CASH_DRAWER_OPEN	String	*true* indicates the cash drawer is open. *false* indicates the drawer is closed.
MSR_TRACK_1	MSR_TRACK_1	String	The value of the track 1 data of the last card swiped through the magnetic stripe reader.
MSR_TRACK_2	MSR_TRACK_2	String	The value of the track 2 data of the last card swiped through the magnetic stripe reader.

Property Name Constant	Property Name	Type	Notes
MSR_TRACK_3	MSR_TRACK_3	String	The value of the track 3 data of the last card swiped through the magnetic stripe reader.
PRINTER_COVER_OPEN	PRINTER_COVER_OPEN	String	*true* indicates the cash receipt printer cover is open. *false* indicates the cover is closed.
PRINTER_DOC_INSERT	PRINTER_DOC_INSERT	String	*true* indicates that there is a document inserted in the printer document insert station. *false* indicates there is no document in the document insert printer station.
PRINTER_OUT_OF_RECEIPT_ PAPER	PRINTER_OUT_OF_RECEIPT_ PAPER	String	*true* indicates the printer is out of receipt paper.
DI_INSERT_DOCUMENT	DI_INSERT_DOCUMENT	String	*true* indicates that there is a prompt to insert a document, *false* if not.
DI_REMOVE_DOCUMENT	DI_REMOVE_DOCUMENT	String	*true* indicates that there is a prompt to remove a document, *false* if not.
LINE_CENTERED	centered	String	*true* if the print line is centered, *false* if not.
LINE_FONT	font	String	Font for the print line.
LOGO_ID	logoId	String	ID of the logo to be printed.
BARCODE_HORIZONTAL	barCodeHorizontal	String	Barcode horizontal scale factor.
BARCODE_VERTICAL	barCodeVertical	String	Barcode vertical scale factor.
BARCODE_HUMAN_READABLE	barCodeHumanReadable	String	Position of the human readable data in relation to the barcode.
BARCODE_TYPE	barCodeType	String	Barcode type.
BARCODE_DATA	barCodeData	String	Barcode data.
FILTERED_DATA	filteredData	String	Formatted print line data.
RAW_DATA	rawData	String	Raw print line data including printer instructions.
LINE_TYPE	lineType	String	Type of print line, such as ItemSale.
COLUMNS_TO_PRINT	columnsToPrint	String	Number of columns that are to be printed.
RECEIPT_COMPLETED	receiptCompleted	String	Indicates the completion of the receipt.

StoreOptionsProperties

The constant StoreOptionsProperties.CATEGORY can be used for the category name. These properties represent the last *store options* configuration values loaded by the POS application.

Property Name Constant	Property Name	Type	**Notes**
NO_SALE_PRICE_VERIFY_IN_ TRANSACTION_ALLOWED	priceVerifyInsideTransaction Allowed	String	*true* indicates that the function is allowed. *false* indicates the function is not allowed. (SA, ACE)
MANAGERS_KEY_NEEDED	overrideRequiresManagerKey	String	*true* indicates that the POS application is configured to require a manager's key for a manager's override. (SA, ACE)
MAXIMUM_TRANSACTION_SIZE	maximumTransactionSize	String	The maximum number of line items allowed in a transaction as configured by the POS application. (SA, ACE)
TRANSACTION_WARNING_SIZE	transactionWarningSize	String	The number of line items configured by the POS application which will generate a transaction size warning. (SA, ACE)
FOOD_STAMPS_ALLOWED	foodstampsAllowed	String	*true* indicates that the function is allowed. *false* indicates the function is not allowed. (SA, ACE)
FOOD_STAMPS_ONLY_ALLOWED	onlyFoodstampsAllowedAfterFood stampTotal	String	*true* indicates that the POS application is configured to accept only foodstamps after a foodstamp total is taken. (SA, ACE)

MAX_SUSPENDED_TRANSACTIONS	maximumSuspendedTransactions	String	The maximum number of suspended transaction as configured in the POS application. (SA, ACE)
CUSTOMER_FUNCTION_CODE	customerFunctionCode	String	They keyboard function code which is configured by the POS application for entry of the customer loyalty number. (SA-EM, ACE)
SUSPEND_TRANSACTION_ALLOWED	suspendTransactionAllowed	String	*true* indicates that the function is allowed. *false* indicates the function is not allowed. (SA, ACE)
WIC_TENDER_ONLY_IN_WIC_TRANS	WICTenderOnlyInWICTransaction	String	*true* indicates that the POS application is configured to allow only WIC tenders within WIC transactions. (SA, ACE)
WEIGHT_DECIMAL_PLACES	weightInputDecimalPlaces	String	The numeric value configured in the POS application for the number of decimal places when entering a weight. (SA, ACE)
FOREIGN_TENDER_SUPPORTED	foreignTenderAllowed	String	*true* indicates that the POS application is configured to accept foreign tenders. (SA, ACE)
ALPHA_STATE_INPUT_ALLOWED	alphaStateInputAllowed	String	*true* indicates that the POS application is configured to allow alphabetic input of states. (ACE)

Data Provider properties

ALPHA_DRIVERS_LICENSE_INPUT_ ALLOWED	alphaDriversLicenseInputAllowed	String	*true* indicates that the POS application is configured to allow alphabetic input of states for driver's license entry. (ACE)
MULTIPLE_CASH_DRAWER_SUPPORT	multipleCashDrawerSupport	String	*true* indicates that the POS application is configured to support multiple cash drawers. (ACE)
OTR_ENABLED	openTransReportEnabled	String	*true* indicates that the open transaction report has been enabled by the POS application. (ACE)
OTR_PRINT_ENABLED	openTransPrintEnabled	String	*true* indicates that printing the open transaction report has been enabled by the POS application. (ACE)
OTR_REFRESH_ENABLED	openTransRefreshEnabled	String	*true* indicates that refreshing the open transaction report has been enabled by the POS application. (ACE)
OTR_FUNCTIONS_ENABLED	openTransFunctionsEnabled	String	*true* indicates that the open transaction report functions have been enabled in the POS application. (ACE)
COUPON_MULTIPLIER_ENABLED	couponMultiplierEnabled	String	*true* indicates that the function has been configured in the POS application. (ACE)

EATIN_TAKEOUT_PROMPT_ENABLED	eatinTakeoutPromptEnabled	String	*true* indicates that the function has been configured in the POS application. (ACE)
VOLUME_INPUT_DECIMAL_PLACES	volumeInputDecimalPlaces	String	The numeric value configured in the application for the number of decimal places used for volume input. (ACE)
VOLUME_UNIT_PRICE_DECIMAL_ PLACES	volumeUnitPriceDecimalPlaces	String	The numeric value configured in the application for the number of decimal places used for volume unit prices. (ACE)
GIFT_RECEIPT_PRINTING_ENABLED	enableGiftReceiptPrinting	String	*true* indicates that the function is configured. (ACE)
PRODUCT_VERSION_ENABLED	enableProductVersion	String	*true* indicates that the product version is allowed to be displayed. (ACE)
SCAN_MANAGER_ID_REQUIRED	scanManagerID	String	*true* indicates that the POS application is configured to require the manager ID to be scanned for manager overrides. (ACE)
WICEBT_ID	WICEBTid	String	WICEBT identifier (ACE)
WICEBT_ENABLED	WICEBTEnabled	String	*true* indicates that WICEBT is enabled, false if not. (ACE)

TRANSACTION_DEFINITION	transactionDefinition	Collection	A Collection of TransactionDefinition elements. Each element represents the configuration settings for a different transaction type. (GSA)
OPER_PERFORMANCE_ENABLED	operPerformanceEnabled	String	*true* indicates that the POS application keeps operator performance metrics.
LOYALTY_ID_CATEGORY	loyaltyID	String	The category of the loyalty ID numbers, for example, 4 indicates IDs in the 4xxxxxxxxxx range.
PROMPT_FOR_VOID_REASON_CODE	promptForVoidReasonCode	String	*true* if the POS application prompts for a void reason code on an item void.
ENHANCED_PASSWORDS_AVAILABLE	enableEnhancedPasswords	String	*true* if the POS application has enhanced passwords enabled. (ACE)
RAIN_CHECK_ENABLED	rainCheckEnabled	String	*true* indicates the POS application has the rain check function enabled. (ACE)
LOCAL_SUSPEND_RETRIEVE_ENABLED	localSuspendRetrieveEnabled	String	*true* indicates that suspending and retrieving to local files, in the case of a controller failure, is supported. (ACE)
VALUE_CARD_SUMMARY_ENABLED	valueCardSummaryEnabled	String	*true* indicates that a value card summary can be requested to print. (ACE)

VALUE_CARD_RECEIPTS_ENABLED	valueCardReceiptsEnabled	String	*true* indicates that value card receipts will be printed. (ACE)

TenderProperties

The constant TenderProperties.CATEGORY can be used for the category name. These properties represent the attributes of the last tender line item in the transaction.

Property Name Constant	Property Name	Type	Notes
TENDER_TYPE	tenderType	String	The type of the tender, as defined by the POS application. For the Toshiba POS applications, the tender type is an integer.
TENDER_VARIETY	tenderVariety	String	The variety of tender, as defined by the POS application. (SA, ACE)
AMOUNT	amount	String	The tender amount, without currency symbols.
ACCOUNT_NUMBER	accountNumber	String	The tender account number if applicable.
EXPIRATION_DATE	expirationDate	String	The tender expiration date if applicable.
FEE	fee	String	The tender fee (without currency symbol) if the POS application applied a fee for taking the tender.
MASKED_ACCOUNT_NUMBER	maskedAccountNumber	String	The masked account number.
RESPONSE_CODE	responseCode	String	The response code from the host.
CHANGE	change	String	The change due in this tender.
IS_BALANCE_SATISFIED	balanceSatisfied	String	*true* indicates that this tender satisfies the balance due.
IS_SIGNATURE_NEEDED	signatureNeeded	String	*true* indicates that this tender requires a customer signature.
IS_REG_E_PRINTED	regEPrinted	String	*true* indicates that a Reg-E receipt is printed for this tender.

TerminalOptionsProperties

The constant TerminalOptionsProperties.CATEGORY can be used for the category name. These properties represent the last options configuration values loaded by the POS application.

Property Name Constant	Property Name	Type	Notes
NO_SALE_ALLOWED	noSaleTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CASH_TRANSACTION_ALLOWED	cashTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
LAYAWAY_ALLOWED	layawayTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)

Data Provider properties

Property Name Constant	Property Name	Type	Notes
RETURNS_ALLOWED	itemReturnTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
VOID_TRANSACTION_ALLOWED	voidTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
TAX_CODE_REQUIRED	taxCodeRequired	String	*true* indicates the POS application is configured to require a tax code when chaning the taxes. (GSA)
OPERATOR_ID_REQUIRED	operatorIDRequired	String	(GSA)
CASH_SPEC_ALLOWED	cashSpecialTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CASH_DOC_ALLOWED	cashDocumentTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CHARGE_PLAN_A_ALLOWED	chargePlanATransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CHARGE_PLAN_B_ALLOWED	chargePlanBTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CHARGE_PLAN_C_ALLOWED	chargePlanCTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
CHARGE_PLAN_D_ALLOWED	chargePlanDTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
COD_ALLOWED	codTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
SEND_ALLOWED	sendTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
PAYMENTS_ALLOWED	paymentsAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)

Property Name Constant	Property Name	Type	Notes
ALLOWANCE_ALLOWED	allowanceAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
DISCOUNTS_ALLOWED	discountsAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)
ORIGINAL_SALES_PERSON_REQUIRED	originalSalesPersonRequired	String	*true* indicates the application is configured to require the entry of the original salesperson. (GSA)
TERMS_OF_SALE_REQUIRED	termsOfSaleRequired	String	(GSA)
SUSPEND_TRANSACTION_ALLOWED	suspendTransactionAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA)

TransactionStatusProperties

The constant TransactionStatusProperties.CATEGORY can be used for the category name. These properties represent information about the POS transaction in progress.

Property Name Constant	Property Name	Type	Notes
ID	id	String	The transaction identifier (number).
DATE	date	String	The date the transaction was initiated.
TIME	time	String	The time the transaction was initiated.
TRANSACTION_CATEGORY	category	String	Should be either "sales" or "nonsales" depending on the transaction type.

Data Provider properties

Property Name Constant	Property Name	Type	Notes
TRANSACTION_TYPE	type	String	The transaction type. Expected values from GSA, SA, & ACE are: • noSale • cash or regularSale • cashSpecial • cashDocument • layaway • cod • chargePlanA • chargePlanB • chargePlanC • chargePlanD • loan • pickup (also known as withdrawal) • tenderCount • totalsReadoutReset • itemMovementRe port • itemPriceChange • setTransactionNumber • offlineReentry • training • voidPreviousTransac tion • voidPreviousByLine Item • tenderListingReport • tenderRemoval • Monitor • queryExchangeRate • tenderCashing • tenderExchange • priceVerify • Transfer • Program Load • itemReturn • wic • reprintPartialReceipt • reprintTenderReceipt • EBTBalanceInquiry • valueCardBalanceInquiry • departmentTotalsReport • layawayPayment • layawayCancel • priceChange • tenderFeeRefund • suspendedTransactionReport • modifyDepartment Presets • openTransactionReport
TAX_REASON	taxReason	String	The tax code or reason code for the last tax change in the transaction.
TAX_TYPE	taxType	String	• (GSA) Set to the type of taxcode. • (SA) Set to "taxExempt" if the taxReason is a taxExempt discount. • (ACE) Set to "taxExempt" if the taxReason is a discount group which exempts all tax plans. Set to "alternateTax" if the taxReason is a discount group which exempts some tax plans

Property Name Constant	Property Name	Type	Notes
TAX_VOIDED	isVoided	String	Set to *true* if the last tax change was voided.
RETURN_PAYMENTS	returnPayments	String	*true* indicates a layaway transaction for returning layaway payments was started. (GSA)
CANCEL_ALL_ITEMS	cancelAllItems	String	*true* indicates ta layaway transaction to cancel all layaway items was started. (GSA)
MODIFIER	modifier	String	Additional transaction information which may be application specific. For example : • return • send • vat • display
ITEM_ALLOWANCE_ALLOWED	itemAllowanceAllowed	String	“true" indicates that item allowance is now valid in the transaction. (GSA)
ITEM_DISCOUNT_ALLOWED	itemDiscountAllowed	String	*true* indicates that item discount is now valid in the transaction. (GSA, ACE)
VOID_LINE_ITEM_ALLOWED	itemVoidAllowed	String	*true* indicates the void line item function is valid in the transaction. (GSA, SA, ACE)
TRANSACTION_DISCOUNT_ALLOWED	transactionDiscountAllowed	String	*true* indicates the transaction discount function is valid in the transaction. (GSA, SA, ACE)
VOID_TRANSACTION_DISCOUNT_ALLOWED	voidTransactionDiscountAllowed	String	*true* indicates the function is allowed, *false* indicates the function is not allowed. (GSA, SA, ACE)
TAX_EXEMPTION_STATUS	taxExemptionStatus	String	*true* indicates the transaction is tax exempt.
TOTAL_PROCESSING_STATUS	totalProcessing	String	Value indicates either the begin or the end of when the transaction totals are being processed.
RETRIEVE_PROCESSING_STATUS	retrieveProcessing	String	Value indicates either the begin or the end of a transaction retrieve process.
LAST_TENDER_APPROVED	lastTenderApproved	String	Set to *false* when a tender action is called, then to *true* when a credit tender is approved.
ITEM_SALE_IN_PROGRESS	itemSaleInProgress	String	Set to *true* when a line item is detected by the AEF, and set to *false* when the AEF receives the Unit of Work message from the POS application.
ACTION_CANCELLED	actionCancelled	String	Value of *true* indicates that the currently executing action has been cancelled. Error helpers can query this to determine how to process an error.
LAST_ITEM_ADDED	lastItemAdded	String	This is set to *false* when an item entry action is called, and then set to *true* if the item is added.

Property Name Constant	Property Name	Type	Notes
EXPECTING_ITEM	expectingItem	String	This is set to *false* when an item entry action is called, and will be set to *true* if an EXPECTING_ITEM or EXPECTING_COUPON substate is received. (ACE)
TENDER_ACTION_CALLED	tenderActionCalled	String	*true* indicates that a tender action has been called.
TENDERING	tendering	String	Set to *true* at the beginning of a tendering call and false at the end of the call.
FINAL_PRICE_PROMPT_CLEARED	finalPricePromptCleared	String	This is set to false when the rain check error helper begins to clear the 'ENTER = FINAL PRICE CLEAR = PER LB PRICE' prompt. It is set to true once the prompt has transitioned to the 'ENTER WEIGHT' prompt. (ACE)

TransactionTotalsProperties

The constant TransactionTotalsProperties.CATEGORY can be used for the category name. These properties reflect the latest running transaction totals for the POS transaction in progress.

Property Name Constant	Property Name	Type	Notes
TOTAL	total	String	The total amount of the transaction.
SUB_TOTAL	subTotal	String	The total transaction amount less any taxes.
TOTAL_SAVINGS	totalSavings	String	The total loyalty savings in the transaction.
TAX	tax	String	The sum of all taxes in the transaction.
AMOUNT_DUE	balanceDue	String	The remaining balance due to satisfy the transaction.
CHANGE_DUE	changeDue	String	The amount of change due to the customer.
FOOD_STAMP_TOTAL	foodStampTotal	String	The total transaction amount which are eligible for food stamps.
FOOD_STAMP_BALANCE	foodStampBalance	String	The remaining food stamps amount due in the transaction.
FOOD_STAMP_CHANGE	foodStampChange	String	The amount of change in food stamps due the customer.
TENDER_EXCHANGED	tenderExchanged	String	The total tender amount (of any type) accepted for this transaction.
COUPON_TOTAL	totalCouponAmount	String	The total amount of coupons tendered in this transaction.
TOTAL_ITEMS	totalItems	String	The total number of items sold in this transaction.
TOTAL_COUPONS	totalCouponCount	String	The total number of coupons tendered in this transaction.
LAYAWAY_BALANCE_DUE	layawayBalanceDue	String	The remaining balance on the layaway account. (GSA)

Property Name Constant	Property Name	Type	Notes
LAYAWAY_FEE	layawayFee	String	For layaway transaction type, set to the layaway fee (if any). For layawayCancel transaction type, set to a negative layaway fee if the fee is refundable. (GSA)
LAYAWAY_DEPOSIT	layawayDeposit	String	For layaway transaction type, set to the layaway deposit amount (if any). For layawayCancel transaction type, set to a negative layaway deposit if the deposit is refundable. For a layawayPayment transaction type, set to the layaway deposit amount (if any). (GSA)
FOREIGN_BALANCE_DUE	foreignBalanceDue	String	The remaining balance translated to the foreign currency as configured in the application. Should be included when a foreign total is taken. (SA, ACE)

WorkstationStatusProperties

The constant WorkstationStatusProperties.CATEGORY can be used for the category name. These properties reflect various attributes of the virtual or real register.

Property Name Constant	Property Name	Type	Notes
_STATUS	Status	String	Values may be WorkstationStatusEvent.DISABLE_TERMINAL_ACKNOWLEDGED or WorkstatoinStatusEvent.INPUT_SEQUENCE_CLEARED.
_NUMBER	_NUMBER	String	Set to the number.
TRANSACTION_NUMBER	TRANSACTION_NUMBER	String	Set to the transaction number of the transaction in progress.
SALES_TRANSACTION_IN_PROGRESS	SALES_TRANSACTION_IN_PROGRESS	String	Set to *true* while a sales transaction is in progress.
TRANSACTION_IN_PROGRESS	TRANSACTION_IN_PROGRESS	String	Set to *true* while a transaction of any type is in progress.
REENTRY_OFFLINE_TRANSACTION	REENTRY_MODE	String	Set to *true* while the register is in offline reentry mode. (GSA)
TRAINING_MODE	TRAINING_MODE	String	Set to *true* while the register is in training mode.

Data Provider properties

Property Name Constant	Property Name	Type	Notes
OFFLINE_MODE	OFFLINE_MODE	String	Set to *true* while the register is offline from its controller. (A virtual CSS session is never offline.)
MESSAGE_PENDING	MESSAGE_PENDING	String	Set to *true* if there is a message pending for the register. On the Toshiba Global Commerce Solutions 4690 OS platform, this is a system message from the OS.
QUEUE_LOCKED	QUEUE_LOCKED	String	Set to *true* if the register's input queue is locked. If the queue is locked, the register cannot accept keyboard or scanner input.
UNIT_OF_WORK	unitOfWork	String	Set to *true* and then to *false* when the POS application has completed processing some input and returned to "idle" state.
OPTIONS_LOADING_IN_PROGRESS	OPTIONS_LOADING_IN_PROGRESS	String	Set to *true* while the register is loading configuration options.
SIGNON_STATUS	signonStatus	String	Indicates whether an operator is signed onto the register. Values are *true*, *false*, or "secureMode".
_DISABLED	Disabled	String	Set to *true* while the has been disabled.

Appendix F. POS application XML events

These XML events are meant only to provide data between the Toshiba Global Commerce Solutions POS applications and the AEF, they are documented here in anticipation that some user extensions might require user data to be included in the events.

Unless otherwise indicated, the term "optional" when applied to XML attributes means that if the underlying POS application does not support the attribute, it doesn't need to be included in the XML event. If the attribute is supported by the underlying POS application then the attribute must always be included in the XML event.

Note: Due to a limitation in SI, POS application XML events cannot be larger than 64 kilobytes in size.

CashReceipt event

cashReceipt

Sent when the POS application writes to the printer.

XML Format

```
<cashReceipt linefeeds="2" paperCut="false">
  <printLine rawData="[\27][\05]VISA #######888   35.23
    filteredData="VISA #######888   35.23"/>
  <printLine rawData="  approval:  1123      "
    filteredData="  approval:  1123    "/>
</cashReceipt>
```

Table 45. <cashReceipt> event

Value	Type	Description
filteredData	String	The same print line as the rawData, except that any printer control characters have been filtered from the string.
linefeeds	String	Indicates the number of lines advanced
paperCut	Boolean	Indicates if a receipt paper cut was performed

Table 45. <cashReceipt> event (continued)

Value	Type	Description
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced. • **signatureLine**(boolean): Indicates whether the print line requires a customer signature For SA, the value of this attribute comes from the JAVA.SIGNATURE.LINE global variable which must be set by the payment system.
rawData	String	Contains the print data including any printer control characters. Characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character.
receiptCompleted	Boolean	Description: Indicates whether or not the current receipt line is the final line for a given receipt type. The final receipt line must be empty.

Coupon event

coupon

Sent when the POS application determines that a coupon has been added to a transaction.

XML Format

```
<Coupon couponType="Store" itemID="80"
  itemIDQualifier="KeyedItemCode" description="Coupon .25"
  value="-.25" isVoided="false" isRefunded="false"
  isDeposit="false" itemRepeatAllowed="false"
  reducesFoodstampBalanceDue="true" reducesTaxDue="true"
  pricingMethod="unit" quantity="1" unitPrice=".25" >
    <printLine rawData="SCCoupon .25 .25-B"
      filteredData="SC Coupon .25.25-B"/>
    <restrict reversed="false" day="sat"
    start="12:00AM" end="11:59AM" />
</Coupon>
```

Note: Toshiba Global Commerce Solutions Electronic Marketing coupons overlay and redefine item record fields to implement various coupon types. These values are shown for completeness, but are implementation details.

Table 46. <coupon> event

Value	Type	Description
couponType	String	Store, Manufacturer, and so on.
dealPrice	String	The price for a deal pricing method. For example, if a deal is 3 for $1.00, the dealPrice is $1.00.
dealQuantity	String	The quantity involved in dealQuantity pricing, or other pricing methods that require a quantity.
departmentNumber	String	The department number associated with the coupon.
description	String	A textual description of the coupon.
enteredPriceUsed	Boolean	Set to true when the price entered was used to sell the coupon.
eventType	String	The type of event.
groupID	String	Identifies the group that the item belongs to for multigroup pricing schemes.
isDeposit	Boolean	Indicates whether the coupon was a deposit. If not voided or refunded, then it was sold.
isRefunded	Boolean	Indicates whether the coupon was refunded. If not voided or refunded, then it was sold.
isVoided	Boolean	Indicates whether the coupon was voided. If not voided, then it was applied to the transaction.
itemID	String	A unique identifier for the coupon. The format of the identifier may be determined by the itemIDQualifier.
itemIDQualifier	String	Indicates the format of the identifier (can be scanned or keyed). For 4690 Supermarket Application and SurePOS Application Client/Server Environment, the values can be: • ScannedItemCode • KeyedItemCode • ItemLookupKeyed • LinkedItemCode • WandedItemCode
itemModifier	String	Optionally gives more information about the type of coupon. This is POS application specific.
itemRepeatAllowed	String	Can this coupon be applied again using a repeat key.
itemTaxable	Boolean	Indicates whether the coupon is taxable.
itemType	String	The type of coupon sold. For example: StoreCouponItem
linkedItemID	String	Item ID linked to this item. (ACE) This attribute is always populated if the item is linked to another item, even if the linked item is not awarded. (SA) This attribute is only populated if the linked item is awarded.
manufacturerNumber	String	The coupon manufacturer number. (Valid only for Manufacturer coupons.)

Table 46. <coupon> event (continued)

Value	Type	Description
pricingMethod	String	The pricing method. For the Supermarket Application and the SurePOS Application Client/Server Environment, the values may be: • unit • basePlus1 • groupThreshold • reducedDeal • increasedDeal • alias For the Supermarket Application the value may also be: • dealQuantity
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
quantity	String	The number of times that the coupon was applied.
reducedPrice	String	The reduced price may be used by certain pricing methods.
reducesFoodstampBalanceDue	Boolean	Indicates whether the coupon reduces the foodstamp balance due.
reducesTaxDue	Boolean	Indicates whether this coupon reduces the tax balance due.
requiredAge	String	The number of years old the customer must be to redeem the coupon.
restrict	Element	Describes any restricted periods for the coupon. Attributes: • reversed="true | false" day="sat | sun | mon | tue | wed | thu | fri" start="HH:MMA M | PM" • end="HH:MM AM | PM"
unitPrice	String	The unit value of the coupon.
value	String	The actual value of the coupon.
weight	String	The weight of this item if sold as a weighted item.
wicEligible	Boolean	Is this coupon eligible for WIC
promotionServiceID	String	The unique ID of the promotion service that added the coupon.
promotionServiceCouponID	String	The unique ID of the coupon added via a promotion service.

Table 46. <coupon> event (continued)

Value	Type	Description
MatchingItem	Element	An element that identifies the item a promotion service coupon matches with. It must contain the following attributes: • ordinalNumber (String): The ordinal number of the matched item • couponValue (String): The value of the coupon
percentageOff	String	The percentage this coupon deducts from the item it matches. Set this only if the coupon isn't a fixed value coupon or a net value coupon.
fixedValueAmount	String	The fixed value of the coupon. Set this only if the coupon isn't a net value coupon or a percentage off coupon.
netValueAmount	String	The net value of the coupon. Set this only if the coupon isn't a fixed value coupon or a percentage off coupon.
promotionServiceUserData	String	An echo of any user data for this coupon sent to the POS application by a promotion service.

DataEvent

Using dataEvent

Sent to set one or more POSDataProvider property values. Provides generic extension mechanism for setting properties with the AEF POSDataProvider.

XML Format

```
<dataEvent dataCategory="xxxxx"
  PROPERTY_NAME1="yyyyy" PROPERTY_NAME2="ZZZZ"/>
```

Table 47. <dataEvent> event

Value	Type	Description
dataCategory	String	Set to one of the following categories: • **POS_DEVICE** (string): Indicates that the data event is associated with a POS Device status change. • **WORKSTATION_STATUS** (string): Indicates that the data event is associated with a workstation status change.
PROPERTY_NAME1	String	*PROPERTY_NAME1* is the name of the property that will have its value set.

The following table provides a list of dataEvents. Data events generate AEFPropertyChangeEvents through the AEF POSDataProvider interface using the dataCategory and PROPERTY_NAME values specified in the xml.

Table 48. <dataEvent> values

Description	AEF XML Message
Options are being loaded into POS terminal sales application, or have completed loading	<dataEvent dataCategory="WORKSTATION_STATUS" OPTIONS_LOADING_IN_PROGRESS="true|false"/>
The substate of the terminal sales application has changed. Substates provide a more granular state change mechanism.	<dataEvent dataCategory="POS_DEVICE" subState="xxxx" />

Table 48. <dataEvent> values (continued)

Description	AEF XML Message
A POS transaction unit of work has started or ended. A unit of work defines a logical beginning and end of a task within a transaction (for example, add an item, or applying a discount).	<dataEvent dataCategory="WORKSTATION_STATUS" unitOfWork="true|false" />
The OPERATOR_DISPLAY property informs the AEF which logical display device is used for prompting the operator (cashier) at the POS terminal.	<dataEvent dataCategory="POS_DEVICE" OPERATOR_DISPLAY="LineDisplay1| LineDisplay2| LineDisplay3"/>
The CUSTOMER_DISPLAY property informs the AEF which logical display device is used for prompting the customer at the POS terminal.	<dataEvent dataCategory="POS_DEVICE" CUSTOMER_DISPLAY=" LineDisplay1| LineDisplay2| LineDisplay3"/>
The terminal is being used in training mode.	<dataEvent dataCategory="WORKSTATION_STATUS" TRAINING_MODE="true|false"/>
The terminal has been disabled (GSA).	<dataEvent dataCategory="WORKSTATION_STATUS" terminalDisabled="true|false"/>
The terminal is being used in transaction reentry mode (GSA).	<dataEvent dataCategory="WORKSTATION_STATUS" REENTRY_MODE="true|false"/>
The terminal sales application needs to write an error (SEVERE) message to the SI log file.	<dataEvent dataCategory="WORKSTATION_STATUS" errorMessage="[The message to write to the SI logs]"/>

Discount events

transactionDiscount

Sent when the POS application applies or voids a discount to a transaction.

XML Format

```
<transactionDiscount description="AARP discount" isVoided="true|false"
  discountMethod="percent|allowance" rate="5.0" amount="0.50"
  discountReason="xx" reducesTaxBalanceDue="true|false" >
    <printLine rawData="   AARP discount -0.50"
      filteredData="  AARP discount -0.50"/>
    </printLine>
</transactionDiscount >
```

Table 49. <transactionDiscount> event

Value	Type	Description
amount	String	The actual amount of the discount. For SA and ACE, include this field only if the discount rate is non-zero.
description	String	A textual description of the discount. The attribute should be provided if the application is able to supply the description.
discountMethod	String	Either percent for a percentage off or allowance for a fixed amount off.
discountReason	String	The discount reason code or discount group (GSA, SA, ACE).
isVoided	Boolean	Indicates whether the discount is being added or voided.

Table 49. <transactionDiscount> event (continued)

Value	Type	Description
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
rate	String	The percentage off for a percent method discount.
reducesTaxBalanceDue	Boolean	Indicates whether the discount reduces the tax balance due. For SA & ACE, include this field only if the discount rate is non-zero.

Note: In the case of a tax exemption, the discountReason can be used to look up the transactionDiscountReason in POSDataProvider to determine the tax exemption information for the discount.

lineItemDiscount

Sent when the POS application applies or voids a line item discount.

XML Format

```
<lineItemDiscount description="AARP discount" isVoided="true|false"
  appliesTo="next" discountMethod="percent|allowance" rate="5.0"
  amount="0.50" discountType="01" discountReason="xxxxxx"
  reducesTaxBalanceDue="true|false">
    <printLine rawData="   AARP discount -0.50"
      filteredData="   AARP discount  -0.50"/>
</lineItemDiscount>
```

Table 50. <lineItemDiscount> event

Value	Type	Description
amount	String	The actual amount of the discount.
appliesTo	String	Indicates the line item the discount applies to. Values may be previous, next or the number of the line item within the transaction (assuming the first line item is item #1).
description	String	A textual description of the discount.
discountMethod	String	Either percent for a percentage off or allowance for a fixed amount off.
discountReason	String	The discount reason code or discount group. (GSA, ACE).
isVoided	Boolean	Indicates whether the discount is being added or voided.

Table 50. <lineItemDiscount> event (continued)

Value	Type	Description
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
rate	String	The percentage off for a percent method discount.
reducesTaxBalanceDue	Boolean	Indicates whether the discount reduces the tax balance due.
Optional data not included in 4690 implementation		
discountType	String	Identifies the type of discount. This may be application specific information.

ItemSales event

itemEvent

Sent when the POS application determines that an item has been added to a transaction. Also sent when an item is voided.

XML Format

```
< itemEvent  attribute1="xxx" attribute2="xxx" >
  <printLine rawData="1  Michelob 40oz.             "
    filteredData="1   Michelob  40oz.             "/>
  <printLine rawData="    3@3.00                  1.00"
    filteredData="     3@3.00            1.00"/>
  <!This is a yearly restrict period, starting Jan 05 at 12:00 AM and
    ending Feb 25 at 11:59 AM
  <restrict reversed="false" startMonth="jan" startDate="5"
    startTime="12:00AM" endMonth="feb" endDate="25"
    endTime="11:59 AM"/>
  <!This is a monthly restrict period, starting on the third day of
    each month at 11:00 PM, and ending on the twentieth day of
    each month at 12:00 PM.
  <restrict reversed="false" startDate="3" startTime="11:00PM"
    endDate="20" endTime="12:00PM"/>
  <!This is a weekly restrict period, starting on Tuesday at 3:34
    PM and ending on Saturday at 6:00 AM each week.
  <restrict reversed="false" startDay="tue" startTime="03:34
    PM" endDay="sat" endTime="06:00AM"/>
  <!This is a daily restrict period, starting at 7:30 AM and
```

```
    ending at 8:59 PM each day.
  <restrict reversed="false" startTime="07:30AM"
    endTime="08:59PM" />
</ itemEvent >
```

Table 51. itemID

Value	Type	Description
dealPrice	String	The price for a deal pricing method. For example, if a deal is 3 for $1.00, the dealPrice is $1.00.
dealQuantity	String	The quantity involved in dealQuantity pricing, or other pricing methods that require a quantity.
departmentNumber	String	The department number associated with the item.
description	String	A textual description of the item.
enteredPriceUsed	String	If a price override or price required item, add enteredPriceUsed="true".
extendedPrice	String	The actual price of the line item.
eventType	String	The type of event. For example, itemSale.
foodstampEligible	Boolean	Indicates whether the item may be purchased via foodstamps.
groupID	String	Identifies the group that the item belongs to for multigroup pricing schemes.
isDeposit	Boolean	Indicates whether the item was a deposit. If not voided or refunded, then it was sold.
isReturn	Boolean	Indicates whether the item was returned. If not voided or returned, then it was sold.
isVoided	Boolean	Indicates whether the item was voided. If not voided or refunded, then it was sold.
itemID	String	A unique item identifier. The format of the item code might be determined by the itemQualifier.
itemIDQualifier	String	Indicates the format of the item code (may be scanned, keyed). For 4690 Supermarket Application, the values can be: • ScannedItemCode • KeyedItemCode • ItemLookupKeyed • LinkedItemCode • WandedItemCode
itemRepeatAllowed	Boolean	Can this item be sold again using a repeat key.
itemTaxable	Boolean	Indicates whether the item is taxable.
itemType	String	The type of item sold. For example, NormalSaleItem.
linkedItemID	String	(ACE) This attribute is always populated if the item is linked to another item, even if the linked item is not awarded. (SA) This attribute is only populated if the linked item is awarded.
linkedItemIDQualifier	String	Item ID qualifier of item linked to this item

Table 51. itemID (continued)

Value	Type	Description
pricingMethod	String	The pricing method. For the Toshiba Supermarket Application and the SurePOS Application Client/Server Environment, the values may be: • unit • basePlus1 • groupThreshold • reducedDeal • increasedDeal • alias For the Supermarket Application the value may also be: • dealQuantity
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
quantity	String	The number of items sold for this line item.
reducedPrice	String	The reduced price may be used by certain pricing methods.
requiredAge	String	The minimum customer age to purchase the item (ACE). **Note:** SA and GSA can support age restricted items only through user extensions.

Table 51. itemID (continued)

Value	Type	Description
restrict	Element	Describes any restricted periods for the item. Include a <restrict> element for each time period. (SA, GSA) Include the restrict periods from options for each time restricted item. SA and GSA allow weekly restricted periods only. (ACE) Include the restrict periods from the group identified in the item record. For ACE, restricted periods may be daily, weekly, monthly, and yearly. Attributes: • **reversed** (boolean): Set to false if the item cannot be sold during this <restrict> time period. Set to true if this item can ***only*** be sold during this <restrict> time period. • **startMonth** (string): The month in which the restrict period begins. Values can be jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec. This attribute is only used for yearly restricted periods (and therefore does not apply to GSA and SA). • **endMonth** (string): The month in which the restrict period ends. Values can be jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec. This attribute is only used for yearly restricted periods (and therefore does not apply to GSA and SA). • **startDate** (string): The numeric day of the month in which the restrict period begins. This attribute is used for yearly and monthly restrict periods (and therefore does not apply to GSA or SA). • **endDate** (string): The numeric day of the month in which the restrict period ends. This attribute is used for yearly and monthly restrict periods (and therefore does not apply to GSA or SA). • **startDay** (string): The day of the week in which the restrict period begins. This attribute is used for weekly restrict periods. Values can be: sun | mon | tue | wed | thu | fri. • **endDay** (string): The day of the week in which the restrict period ends. This attribute is used for weekly restrict periods. Values can be: sun | mon | tue | wed | thu | fri. • **startTime**: The start of the restrict time period in HH:MMAM | PM format. • **endTime** (string): The end of the restrict time period in HH:MMAM | PM format.
returnReason	String	If the isReturn attribute was set to "true", the field should be included if there is a return reason code. The value should be the return reason code. (ACE)
unitPrice	String	The unit price of the item.
weight	String	The weight of this item if sold as a weighted item.
wicEligible	Boolean	Indicates whether the item may be purchased via a WIC voucher (SA, ACE).
itemOrdinalNumber	String	A unique number that identifies an item's position in the transaction
voidedItemOrdinalNumber	String	This attribute is only valid for voided items. It indicates the ordinal number of the item the voided item negates.

Table 51. itemID (continued)

Value	Type	Description
QuantityEntry	Element	An element used for multi-quantity entries to specify the price and order of each individual item. The attributes for this element are: • ordinalNumber (String): The ordinal number of the individual entry • voidedItemOrdinalNumber (String): Only valid on voids. Indicates the ordinal number of the item this entry negates • unitPrice (String): The unit price of the entry
isDiscountable	Boolean	Indicates whether coupons can apply to this item or not.
promotionCode	String	The promotion code of the item.
promotionServiceGiftCardID	String	The ID of the promotion service which awarded the gift card.
promotionServiceUserData	String	An echo of any user data for this coupon sent to the POS application by a promotion service.
promotionServiceID	String	The unique ID of the promotion service which awarded this item.
fixedValueAmount	String	The fixed value of the item if it is a promotion service gift card.
barcode	String	The barcode for the item. Only populate this field if the barcode is different than the item lookup key.
barcodeFamily	String	The barcode Family. It should contain one of the following case sensitive values: • UPC • EAN • GS1 DataBar • GS1 DataBar Expanded • Other
checkDigit	Boolean	Indicates whether the barcode includes a check digit.
rainCheckKeyed	Boolean	Indicates whether the operator keyed in a rain check price for the item.
weightOrigin	String	Indicates which device the weight originated from. This should be one of the following case sensitive values: • KEYED - The weight was keyed by the operator. • SCANNED - The weight originated from the scale. • CALCULATED - The weight was either embedded in the barcode or was calculated based on some other piece of data.
enteredQuantityUsed	Boolean	Indicates whether the quantity was entered through the keybaord by the operator or not.
depositKeyUsed	Boolean	Indicates whether the deposit key was used to sell an item or not.

Operator events

signOn

Sent when the operator logs onto the POS application.

XML Format

```
<signOn operatorID="xxxx" operatorName="yyyyyy yyyy">
  <authorization auth_prop1="true"  auth_prop2="false" />
</signOn>
```

Table 52. <signOn> event

Value	Type	Description
authorization	Element	Authorization properties are listed as attributes. • **auth_propN** (boolean): Each authorization property is listed by name in the signon event, along with a boolean indicating whether the operator is authorized to perform the function represented by the property name. Authorization properties for AEF enabled Toshiba applications are provided in the interface class OperatorAuthorizationProperties. See Table 53 for a list of the authorization property names for each application.
operatorID	String	The Id of the operator.
operatorName	String	The name of the operator.

This table lists the operator authorization attributes for each application that might be expected in the <signOn> event.

Table 53. Operator authorization attributes

Operator Authorization Property Name	GSA	SA	ACE
allowanceAllowed	X		
cashDocumentTransactionAllowed	X		
cashSpecialTransactionAllowed	X		
cashTransactionAllowed	X		
chargeType6TransactionAllowed	X		
chargeType7TransactionAllowed	X		
chargeType8TransactionAllowed	X		
chargeType9TransactionAllowed	X		
codTransactionAllowed	X		
delayedItemChangeAllowed		X	X
delayedPriceChangeAllowed	X		
deptTotalsReportAllowed	X	X	X
discountsAllowed	X	X	X
frontEndCashierAllowed		X	X
immediateItemChangeAllowed		X	X
itemPriceChangeAllowed	X		
layawayAllowed	X		
loanAllowed	X	X	X
managersProceduresAllowed		X	X
miscItemPayoutAllowed		X	X
moreThanPriceChangesAllowed		X	X
nosaleOpenCashDrawerAllowed	X	X	X
nosalePriceVerifyAllowed		X	X
nosaleTenderRemovalAllowed		X	X
nosaleTenderVerifyAllowed		X	X
nosaleTillExchangeAllowed		X	X
nosaleTillReportAllowed		X	X

Table 53. Operator authorization attributes (continued)

Operator Authorization Property Name	GSA	SA	ACE
obtainItemMovementAllowed	X		
operatorTrainingAllowed	X	X	X
passwordRequired	X		
pickupAllowed	X (known as withdrawals)	X	X
priceVerifyAllowed	X	X	
queryExchangeRateAllowed	X		
reentryOfflineSalesAllowed	X		
registerReadoutAllowed	X		
registerResetAllowed	X		
reprintTenderReceiptAllowed		X	X
returnsAllowed	X	X (known as refunds)	X (known as refunds)
returnItemAllowed		X	X
salesAllowed		X	X
setTransactionNumberAllowed	X		
suspendTransactionAllowed	X		
tenderCashingAllowed		X	X
tenderCountAllowed	X (known as cash count)	X	X
tenderExchangeAllowed		X	X
tenderListingAllowed	X	X	X
tenderRemovalAllowed	X		
terminalMonitorAllowed	X	X	X
terminalTransferAllowed		X	X
tplTransactionAllowed	X		
userFunction1Allowed		X	X
userNonsales1Allowed		X	X
userNonsales2Allowed		X	X
voidPreviousItemAllowed	X		
voidPreviousTransactionAllowed	X		
voidTransactionAllowed	X		
wicTransactionAllowed		X	X

signOff

Sent when the operator logs off the POS application.

XML Format

```
<signOff operatorID="xxxx"/>
```

Table 54. <signOff> event

Value	Type	Description
operatorID	String	The operator ID

lock

Sent when operator locks the workstation, or enters secured mode.

XML Format

```
<lock operatorID="xxxx"/>
```

Table 55. <lock> event

Value	Type	Description
operatorID	String	The operator ID

unlock

Sent when operator unlocks the workstation by entering password.

XML Format

```
<unlock operatorID="xxxx"/>
```

Table 56. <unlock> event

Value	Type	Description
operatorID	String	The operator ID

Customer event

Customer

Sent when the POS application has identified a loyalty customer, possibly by scanning, or swiping a loyalty card.

XML Format

```
<customer id="90071234566" loyaltyIDType="primary|alternate"
  expirationDate="MMYY" name="Carina Lopez"
  email="carina@us.ibm.com">
    <targetedCoupon couponCode="0312" />
    .
    .
    .
    <pointsTotal type="primary" description="Primary Points Total"
      points="0" />
    <pointsBalance type="primary" description="Primary Points Balance"
      points="0" />
    <message>  VALUED CUSTOMER   Carina Lopez  </message>
</customer>
```

Note: The POS application may optionally include one or more targeted coupons for the customer. Each is identified by a coupon-code. The POS application may also include a message specifically for the customer. There may also be one or more <pointsBalance> and <pointsTotal> elements included.

Table 57. <customer> event

Value	Type	Description
email	String	The customer's email address. **Note:** The Supermarket application leaves this field blank.

Table 57. <customer> event (continued)

Value	Type	Description
exprirationDate	String	An optional field that is provided when an expiration date was provided with the loyalty id.
id	String	The customer loyalty number.
loyaltyIDType	String	Either 'primary' or 'alternate'. Example of an alternate loyalty id is the customer's phone number.
messages	String	The message displayed by the application when the customer is recognized.
name	String	The customer's name in the application specific format.
pointsBalance	Element	The remaining points after subtracting any points that have been redeemed by the customer. The type attribute is used to distinguish points of different types. Each <pointsBalance> element contains the following attributes: • **type** (string): An indicator of the type of points. For example, a club. • **description** (string): A description of the points. For example, the club name. • **points** (string): The points balance of the specific type for the customer.
pointsTotal	Element	The total number of points (as determined by the POS application) for the customer. The type attribute is used to distinguish points of different types. Each <pointsTotal> element contains the following attributes: • **type** (string): An indicator of the type of points. For example, a club. • **description** (string): A description of the points. For example, the club name. • **points** (string): The number of points of the specific type for the customer.
targetedCoupon	Element	For each coupon targeted to the customer, include a <targetedCoupon> element with the following attributes: • **couponCode** (string): The item code of the coupon.
Optional attributes supported by the CustomerEvent, but not provided by Toshiba Global Commerce Solutions 4690 applications.		
firstName	String	The customer's first name only
lastName	String	The customer's last name only
address1	String	The customer address line 1
address2	String	The customer address line 2
city	String	The customer's city
state	String	The customer's state
zip	String	The customer zip code
phone	String	The customer phone number
contact	String	The customer contact person
fax	String	The customer's fax number
ytdPoints	String	The customer's points total year-to-date
ytdSaved	String	The customer's savings year-to-date

Table 57. <customer> event (continued)

Value	Type	Description
customerIDType	String	The type of customer identifier in this event. The applicable values are: • LOYALTY_ID

Options event

Options

Sent when the POS terminal sales application loads or modifies options or configuration data that specify the behavior of the POS application. This is also sent upon initialization of the POSBC.

XML Format

```
<LocaleOptions decimalSeparator="." groupingSeparator="," amountDecimalPlaces="2">
<!- Logo definitions (0 - N allowed) -->
<logoDefinition logold="1" logoName="aLogoName" />
<options>
   <store name="Raleigh Store #1201" number="1201"
    phone="919-555-1212" />
   <!— Price override reason codes (0 to N allowed)) —>
   <priceOverrideReason description="desc1" code="code1"/>
.
.
   <priceOverrideReason description="descN" code="codeN"/>
   <!— Refund reason codes (0 to N allowed) —>
   <refundReason description="desc1" code="code1"/>
.
.
   <refundReason description="descN" code="codeN"/>
   <!— Void reason codes (0 to N allowed) —>
   <voidReason description="desc1" code="code1"/>
   <voidReason description="descN" code="codeN"/>
   <!— Tare codes (0 to N allowed) —>
   <tareCode description="desc1" code="code1"/>
.
.
   <tareCode description="descN" code="codeN"/>
   <!— Transaction discount reason codes (0 to N allowed) —>
   <transactionDiscountReason description="xxx"
   code="nnn" rate="zz.z"
              taxExemption="true|false"
              taxPlan1Exempt="true|false"
              taxPlan2Exempt="true|false"
              taxPlan3Exempt="true|false"
              taxPlan4Exempt="true|false"
              taxPlan5Exempt="true|false"
              taxPlan6Exempt="true|false"
              taxPlan7Exempt="true|false"
              taxPlan8Exempt="true|false"/>
.
.
   <!— Item discount reason codes (0 to N allowed) —>
   <itemDiscountReason description="xxx" code="nnn" rate="zz.z"/>
   <!Tender definitions (0 to N allowed) —>
   <tenderDefinition description="CASH" shortDescription="CSH"
     type="01" variety="01" key="xxx" foreignTender="true|false"
     allowed="true|false" canBeLoaned="true|false"
     canBePickedUp="true|false" canBeCounted="true|false"
     canBeRefunded="true|false" canBeVerified="true|false"
   tenderExchangeRank="05" />
.
.
```

```
   <!VAT Tax Codes (0 to N allowed) –>
   <vatTaxCode code="xx" percent="nn.n"/>
.
.
<storeOptions alphaStateInputAllowed="true|false"
 alphaDriversLicenseInputAllowed="true|false"
 priceVerifyInsideTransactionAllowed="true|false"
 multipleCashDrawerSupport="true|false"
 overrideRequiresManagerKey="true|false"
 maximumTransactionSize="nn"
 transactionWarningSize="nn"
 foodstampsAllowed="true|false"
 onlyFoodstampsAllowedAfterFoodstampTotal="true|false"
 maximumSuspendedTransactions="nn"
 customerFunctionCode="nn"
suspendTransactionAllowed="true|false"
 WICTenderOnlyInWICTransaction="true|false"
 weightInputDecimalPlaces="x"
 volumeInputDecimalPlaces="3"
 volumeUnitPriceDecimalPlaces="3"
 enableGiftReceiptPrinting ="true|false"
 foreignTenderAllowed="true|false"
 openTransReportEnabled="true|false"
 openTransPrintEnabled="true|false"
 openTransRefreshEnabled="true|false"
 openTransFunctionsEnabled="true|false"
 couponMultiplierEnabled="true|false"
 scanManagerID="true|false"
 eatinTakeoutPromptEnabled="true|false">

<transactionDefinition transactionType="noSale"
  promptForAccountNumber="true|false"
 documentInsertUsed="true|false"
 paymentsAllowed="true|false"
 allowancesAllowed="true|false"
 discountsAllowed="true|false"
 returnsAllowed="true|false"
 promptForOriginalSalesperson="true|false"
 promptForTermsOfSale="true|false"
 voidTransactionAllowed="true|false"
 suspendTransactionAllowed="true|false"/>
<transactionDefinition transactionType="regularSale"
 promptForAccountNumber="true|false"
 documentInsertUsed="true|false"
 paymentsAllowed="true|false"
 allowancesAllowed="true|false"
 discountsAllowed="true|false"
 returnsAllowed="true|false"
 promptForOriginalSalesperson="true|false"
 promptForTermsOfSale="true|false"
 voidTransactionAllowed="true|false"
 suspendTransactionAllowed="true|false"/>

<transactionDefinition transactionType="cashSpecial"  />
<transactionDefinition transactionType="cashDocument"  />
<transactionDefinition transactionType="cod"  />
<transactionDefinition transactionType="layaway"  />
<transactionDefinition transactionType="chargePlanA"  />
<transactionDefinition transactionType="chargePlanB"  />
<transactionDefinition transactionType="chargePlanC"  />
<transactionDefinition transactionType="chargePlanD" />
</storeOptions>
.
.
.
<terminalOptions noSaleTransactionAllowed="true|false"
  cashTransactionAllowed="true|false"
  layawayTransactionAllowed="true|false"
```

```
  itemReturnTransactionAllowed="true|false"
  voidTransactionAllowed="true|false"
  taxCodeRequired="true|false"
  operatorIDRequired="true|false"
  cashSpecialTransactionAllowed="true|false"
  cashDocumentTransactionAllowed="true|false"
  chargePlanATransactionAllowed="true|false"
  chargePlanBTransactionAllowed="true|false"
  chargePlanCTransactionAllowed="true|false"
  chargePlanDTransactionAllowed="true|false"
  codTransactionAllowed="true|false"
  sendTransactionAllowed="true|false"
  paymentsAllowed="true|false"
  allowanceAllowed="true|false"
  discountsAllowed="true|false"
  originalSalesPersonRequired="true|false"
  termsOfSaleRequired="true|false"
  suspendTransactionAllowed="true|false" >
</terminalOptions>
<!– Department definitions (0 to N allowed) –>
<departmentDefinition
  description="aaaaaa"
  number="xxxx"
  key="nn"
  toggleTaxAllowed="true|false|managerOverride|operatorOverride"
  toggleFoodstampAllowed="true|false|managerOverride|
  operatorOverride"
  refundAllowed="true|false|managerOverride|operatorOverride"
  storeCouponAllowed="true|false|managerOverride|operatorOverride"
  manufacturerCouponAllowed="true|false|managerOverride|
  operatorOverride"
  priceAllowed="true|false|managerOverride|operatorOverride" />
.
.
   <!Alternate Tax codes (0 to N allowed) –>
   <alternateTaxCode code="xx" description="yyyyyy"/>
.
.
   <! Manual Tax codes (0 to N allowed) –>
   <manualTaxCode code="xx" description="yyyyyy"/>
.
.
   <! No Tax codes (0 to N allowed) –>
   <noTaxCode code="xx" description="yyyyyy"/>
.
.
</options>
```

Table 58. <options> event

Value	Type	Descriptions
alternateTaxCode	Element	(GSA) Include a separate <alternateTaxCode> element for each alternate tax code defined in the application. (ACE) Include a separate <alternateTaxCode> for each discount group with a zero discount rate, and that does not exempt all the tax plans. Each <alternateTaxCode> element includes the following attributes: • **description** (string): A textual description of the tax code or discount group, for example, "ILL state tax only". Include this attribute if the application is able to provide the data. • **code** (string): (GSA) The tax code. (ACE) The discount group.

Table 58. <options> event (continued)

Value	Type	Descriptions
departmentDefinition	Element	Include a separate <departmentDefinition> element for each department defined in the application. Each <departmentDefinition> includes the following attributes: • **description** (string): A textual description of the department, for example, "Meat". Include this attribute if the application is able to provide the data. • **number** (string): The department number. • **key** (string): The function code on the keyboard which is used to ring sales for this department. • **toggleTaxAllowed** (string): (SA) Values are "true" or "false". (ACE) Values are "true", "false", "managerOverride", or "operatorOverride". • **toggleFoodstampAllowed** (string): (SA) Values are "true" or "false". (ACE) Values are "true", "false", "managerOverride", or "operatorOverride". • **refundAllowed** (string): (SA) Values are "true" or "false". (ACE) Values are "true", "false", "managerOverride", or "operatorOverride". • **storeCouponAllowed** (string): (SA) Values are "true" or "false". (ACE) Values are "true", "false", "managerOverride", or "operatorOverride". • **manufacturerCouponAllowed** (string): (SA) Values are "true" or "false". (ACE) Values are "true", "false", "managerOverride", or "operatorOverride". • **priceAllowed** (string): (ACE) Values are "true", "false", "managerOverride", or "operatorOverride".
itemDiscountReason	Element	(GSA, ACE). Include a separate <itemDiscountReason> element for each item discount reason or group. In ACE, only discount groups that do not include tax exemptions may be used as item discounts. Therefore, no discount groups that include tax exemptions should be included as an item discount reason. Each <itemDiscountReason> includes the following attributes: • **description** (string): Optional textual description of the discount reason. For example, medium meat tray. Include if the application is able to provide the description. • **code** (string): The numeric discount reason. For ACE, this is the discount group. • **rate** (string): The discount percentage (for example, 10.0 for a ten percent discount.) (ACE)
localeOptions	Element	(SA, ACE) An optional event that specifies number formatting options configured in the Point Of Sale. This element includes the following attributes: • **decimalSeparator** (string): The symbol used to mark the boundary between the integral and the fractional parts of a decimal numeral. • **groupingSeparator** (string): The symbol used to separate numbers to the left of the decimal separator into groups of three. • **amountDecimalPlaces** (string): The number of digits that will appear to the right of the decimal separator.
manualTaxCode	Element	(GSA) Include a separate <manualTaxCode> element for each manual tax code defined in the application. Each <manualTaCodex> element includes the following attributes: • **description** (string): A textual description of the tax code. Include this attribute if the application is able to provide the data. • **code** (string): The tax code.

Table 58. <options> event (continued)

Value	Type	Descriptions
noTaxCode	Element	(GSA) Include a separate <noTaxCode> element for each zero tax code defined in the application. (SA) Include a separate <noTaxCode> element for each discount group with a discount rate of zero, and taxExemption set true. (ACE) Include a separate <noTaxCode> element for each discount group with a discount rate of zero, and tax exemptions of all tax plans. Each <noTaxCode> element includes the following attributes: • **description** (string): A textual description of the tax code or discount group. Include this attribute if the application is able to provide the data. • **code** (string): (GSA) The tax code. (SA,ACE) The discount group.
priceOverrideReason	Element	(ACE). Include a separate <refundReason> element for each price override reason code. Each <refundReason> includes the following attributes: • **description** (string): Optional textual description of the discount reason. For example, "employee discount". Include if the application is able to provide the description. • **code** (string): The price override reason code.
refundReason	Element	(ACE). Include a separate <refundReason> element for each refund reason code. Each <refundReason> includes the following attributes: • **description** (string): Optional text description of the discount reason. For example, spoiled produce. Include if the application is able to provide the description. • **code** (string): The refund reason code.
store	Element	General information about the store (include those attributes for which the application has data) • **Name** (string): Store name • **Number** (string): Store number • **Division** (string): Division • **Address1** (string): Address line 1 • **Address2** (string): Address line 2 • **City** (string): City • **State** (string): State • **zip** (string): Zip code • **phone1** (string): Phone Number 1 • **phone2** (string): Phone Number 2

Table 58. <options> event (continued)

Value	Type	Descriptions
storeOptions	Element	The <storeOptions> element includes the following attributes: • **alphaStateInputAllowed** (boolean) - Indicates whether alphabetic input is allowed for state input (ACE). • **alphaDriversLicenseInputAllowed** - Indicates whether alphabetic input is allowed for drivers license input (ACE). • **priceVerifyInsideTransactionAllowed** (boolean) - Indicates whether the operator may perform a price verification procedure inside a sales transaction. (SA, ACE) • **overrideRequiresManagerKey** (boolean) - Indicates if the manager key is required to complete a manager override. (SA, ACE) • **maximumTransactionSize** (string) - The maximum number of items the application allows within a single transaction. (SA, ACE) • **transactionWarningSize** (string) - The number of items at which a transaction size warning is generated. (SA, ACE) • **foodstampsAllowed** (boolean) - Indicates whether foodstamps are accepted in the store. (SA, ACE) • **onlyFoodstampsAllowedAfterFoodstampTotal** (boolean) - Indicates whether foodstamps must be used after a foodstamp total. (SA, ACE) • **maximumSuspendedTransactions** (string) - The maximum number of transactions allowed to be suspended per terminal or operator. (SA, ACE) • **customerFunctionCode** (string) - The keyboard function code used for customer loyalty number entry. (SA-EM, ACE) • **suspendTransactionAllowed** (boolean) - Indicates whether suspending transactions is allowed in the store. (SA, ACE) • **WICTenderOnlyInWICTransaction** (boolean) - Indicates whether WIC is the only tender allowed in a WIC transaction. (SA, ACE) • **weightInputDecimalPlaces** (string) - The number of significant digits after the decimal for keyboard weight input. (SA, ACE) • **openTransReportEnabled** (ACE) • **openTransPrintEnabled** (boolean)(ACE) • **openTransRefreshEnabled** (boolean)(ACE) • **openTransFunctionsEnabled** (boolean)(ACE) • **couponMultiplierEnabled** (boolean)(ACE) • **eatinTakeoutPromptEnabled** (boolean)(ACE) • **scanManagerID** (boolean)(ACE) • **foreignTenderAllowed** (boolean) - Indicates whether foreign tender is allowed in the store. (SA, ACE) • **transactionDefinition** (element)(GSA) - For each of the 9 transaction types, include a <transactionDefinition> element which lists the transactionType and attributes for the 10 functions indicating whether they are allowed or disallowed. • **volumeInputDecimalPlaces** (string) - The number of significant digits for volume input (ACE) • **volumeUnitPriceDecimalPlaces** (string) - The number of significant digits for pricing by volume (ACE) • **enableGiftReceiptPrinting** (boolean) - Enable the printing of gift receipts. (ACE) • **loyaltyIDCategory** (string) - The category (first digit) of the store's loyalty numbers. (SA, ACE) • **WICEBTEnabled** (boolean) - Indicates whether the WIC EBT function is enabled or not.
tareCode	Element	(SA, ACE). Includes a separate <tareCode> element for each tare code. Each <tareCode> includes the following attributes: • **description** (string): Optional textual description of the tare reason. For example, medium meat tray. Include if the application is able to provide the description. • **Code** (string): The numeric tare code.

Table 58. <options> event (continued)

Value	Type	Descriptions
tenderDefinition	Element	(GSA, SA, ACE). Include a separate <tenderDefinition> element for each tender in the application. Each <tenderDefinition> includes the following attributes: • **description** (string): Textual description of the tender. For example, CASH. (GSA, SA, ACE) • **shortDescription** (string): Provided only if the app has short descriptions for the tender. In GSA, this is the Tender Foreign Currency 2 (or 3) currency identifier in store options for the tender. For GSA, only foreign tenders have a shortDescription. For ACE, this is the Short Description in the tender definition. SA does not support short descriptions for tenders. • **type** (string): The tender type as defined by the application. For example, 4 (GSA, SA, ACE). • **variety** (string): The tender variety as defined by the application. For example, 2 (SA, ACE). • **key** (string): The function code of the tender on the keyboard (GSA). • **foreignTender** (boolean): Indicates if the tender is a foreign tender (GSA, SA, ACE). • **allowed** (boolean): Indicates if the register accepts this tender (GSA, SA, ACE). • **canBeLoaned** (boolean): For SA set to true. For ACE, set according to application definition. • **canBePickedUp** (boolean): (SA) Set true. (ACE) Set according to application definition. • **canBeCounted** (boolean): (SA) Set true. (ACE) Set according to application definition. • **canBeRefunded** (boolean): (SA,ACE) Set according to application definition. • **canBeVerified** (boolean): (SA,ACE) Set according to application definition. • **tenderExchangeRank** (string): (SA,ACE) Set according to application definition.
terminalOptions	Element	The <terminalOptions> element includes the following attributes: • noSaleTransactionAllowed (boolean)(GSA) • cashTransactionAllowed (boolean)(GSA) • layawayTransactionAllowed (boolean)(GSA) • itemReturnTransactionAllowed (boolean)(GSA) • voidTransactionAllowed (boolean)(GSA) • taxCodeRequired (boolean)(GSA) • operatorIDRequired (boolean)(GSA) • cashSpecialTransactionAllowed (boolean)(GSA) • cashDocumentTransactionAllowed (boolean)(GSA) • chargePlanATransactionAllowed (boolean)(GSA) • chargePlanBTransactionAllowed (boolean)(GSA) • chargePlanCTransactionAllowed (boolean)(GSA) • chargePlanDTransactionAllowed (boolean)(GSA) • codTransactionAllowed (boolean)(GSA) • sendTransactionAllowed (boolean)(GSA) • paymentsAllowed (boolean)(GSA) • allowanceAllowed (boolean)(GSA) • discountsAllowed (boolean)(GSA) • originalSalesPersonRequired (boolean)(GSA) • termsOfSaleRequired (boolean)(GSA) • suspendTransactionAllowed (boolean)(GSA)

Table 58. <options> event (continued)

Value	Type	Descriptions
transactionDefinition	Element	(GSA) The <transactionDefinition> element is included for each of the following transaction types: noSale, regularSale, cashSpecial, cashDocument, cod, layaway, chargePlanA, chargePlanB, chargePlanC, chargePlanD. It includes the following attributes: • **transactionType** (string): One of the nine possible values. • **promptForAccountNumber** (boolean): As set in GSA options. • **documentInsertUsed** (boolean): As set in GSA options. • **paymentsAllowed** (boolean): As set in GSA options. • **allowancesAllowed** (boolean): As set in GSA options. • **discountsAllowed** (boolean): As set in GSA options. • **returnsAllowed** (boolean): As set in GSA options. • **promptForOriginalSalesperson** (boolean): As set in GSA options. • **promptForTermsOfSale** (boolean): As set in GSA options. • **voidTransactionAllowed** (boolean): As set in GSA options. • **suspendTransactionAllowed** (boolean): As set in GSA options.
transactionDiscountReaon	Element	(GSA, SA, ACE). Include a separate <transactionDiscountReason> element for each transaction discount reason or group. Each <transactionDiscountReason> includes the following attributes: • **description** (string): Optional textual description of the discount reason. For example, "medium meat tray". Include if the application is able to provide the description. • **code** (string): The numeric discount reason. For SA and ACE, this is the discount group. • **rate** (string): The discount percentage (for example, 10.0 for a ten percent discount. (SA, ACE). • **taxExemption** (boolean): (SA) Set to true if the discount group is a tax exemption (ACE). Set to true if the discount group exempts all active tax plans. • **taxPlan1Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 1. • **taxPlan2Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 2. • **taxPlan3Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 3. • **taxPlan4Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 4. • **taxPlan5Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 5. • **taxPlan6Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 6. • **taxPlan7Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 7. • **taxPlan8Exempt** (boolean): (ACE) Set to true if the discount group is set to exempt tax plan 8.
vatTaxCode	Element	(GSA). Include a separate <vatTaxCode> element for each VAT tax code in the application. Each <vatTaxCode> includes the following attributes: • **code** (string): The numeric tax code. • **Percent** (string): The tax percentage (for example, "10.0" for a ten percent tax.)
logoDefinition	Element	(ACE). An optional event that specifies the mapping between the printer pre-load index of a logo and the URL for the logo: • **logoId** (string): The index of the pre-loaded logo image in the printer. • **logoName** (string): The uniform resource identifier of the logo. The value can be anything representable by a URL, including a URL or a file name.

Points event

Points

Sent when the customer has redeemed or been awarded loyalty points within a transaction.

XML Format

```
<points id="xxxx" idQualifier="xxxx" quantity="1" type="primary"
  description="Bonus Points" points="1000" redeemed="true|false"
  isVoided="true|false">
    <pointsTotal type="club1" description="Gold Club Points Total"
      points="1000" />
    <pointsTotal type="club2" description="Baby Club Points Total"
      points="1000" />
    <printLine rawData="[27][5]     *** Super Saver Bonus ***   "
       filteredData="        *** Super Saver Bonus ***  "/>
    <printLine rawData="     + 1000 Points         "
         filteredData="     + 1000 Points         "/>
</points>
```

Table 59. <points> event

Value	Type	Description
description	String	The description of the points reward.
id	String	A unique identifier of the points reward.
idQualifier	String	Indicates format of the points identifier.
isVoided	Boolean	Indicates whether the points have been voided.
points	String	The total points for the primary account.
pointsTotal	Element	The type, description, and value of the points redeemed or awarded for clubs or secondary accounts.
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
quantity	String	The number of points items sold for this line item.
redeemed	Boolean	True if the points are redeemed, false if they are awarded.
type	String	The type of points.

Report event

report

Sent when the POS application has report information available.

Note: ACE uses this event for OTR.

XML Format

```
<report id="OTR">
  <section id="1">
     <line type"0" data="TERMINAL   DATE   TIME  AMOUNT TRANS." />
  </section>
  <section id="2">
     <line type="0" data="000 7900      9093   9022." />
     <line type="0" data="001 8809      9093   9022." />
   </section>
</report>
```

Table 60. <report> event

Value	Type	Description
id	String	Report identifier
line	element	• **type** (string): Identifies the type of line or attributes of the line to be displayed. • **data** (string): The data to be displayed.
section	element	• **id** (string): Identifies a section of the report or a function to be performed.

Scale event

scale

Sent when the POS application reads the scale for a weight item.

XML Format

```
<scale scaleWeightValue="2.67" scaleWeightUnit="LB"
  scaleWeightLabels="ACE SurePOS" />
```

Table 61. <scale> event

Value	Type	Description
scaleWeightLabels	String	Optional data. Used for displaying weights and measures information.
scaleWeightUnit	String	Weight unit of measure.
scaleWeightValue	String	Weight of the item as a formatted string.

Tender event

Tender

Sent when the POS application determines that a credit tender has been added or voided. See “Tender or tenderDeclined” on page 261 for Value, Type, and Description.

XML Format

```
<tender tenderType="xxx" tenderVariety="yyy" isVoided="true|false"
  description="VISA" amount="35.23" currency="USD" fee="0.50"
  accountNumber="3893828382" referenceNumber="121665">
```

```
    <printLine rawData="VISA #######888  35.23"
      filteredData="VISA #######888    35.23"/>
    <printLine rawData="      3893828382         "
       filteredData="      3893828382     "/>
</tender>
```

Tender declined

Sent when the POS application determines that a credit tender has been declined. See "Tender or tenderDeclined" for Value, Type, and Description.

XML Format

```
<tenderDeclined tenderType="xxx" tenderVariety="yyy" isVoided="true|false"
 description="VISA" amount="35.23" currency="USD" fee="0.50"
 accountNumber="3893828382" referenceNumber="121665">
  <printLine rawData="VISA #######888 35.23"
   filteredData="VISA #######888 35.23"/>
  <printLine rawData=" 3893828382 "
   filteredData=" 3893828382 "/>
</tender>
```

Tender or tenderDeclined

Table 62. <tender> or <tenderDeclined> event

Value	Type	Description
accountNumber	String	The account number of the tender.
amount	String	The amount of the tender.
approvalCode	String	The approval code for the tender. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.APPROVAL.CODE$.
balanceSatisfied	Boolean	Indicates whether or not the tender satisfies the entire balance and causes the transaction to end at the Point Of Sale.
cardID	String	Optionally, used for credit tenders when the entry of the tender requires an additional card ID to be input. This is usually an extension of the account number printed on the back of the card.
change	String	The amount of change due to the customer.
currency	String	The currency of the tender as defined by the three-character country code in ISO 4217. See www.bsi-global.com/iso4217currency. **Note:** The Toshiba 4690 POS applications do not provide this data.
description	String	A textual description of the tender.
expirationDate	String	Used for credit tenders; MMYY format.
fee	String	Any fee charged for accepting the tender.
isVoided	String	A flag that tells us if the tender was voided.
legal	Element	An element containing the legal text associated with the tender. The attributes of this element are: • **text** (string): The printed text with the printer control characters filtered out. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.LEGAL.TEXT$.
maskedAccountNumber	String	The account number obfuscated to meet PCI compliance. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.MASKED.ACCOUNT.NUMBER$.

Table 62. <tender> or <tenderDeclined> event (continued)

Value	Type	Description
printLine	Element	Includes the raw and filtered print lines that are sent to the POS receipt for this event. • **rawData** (string): Contains the print data including any printer control characters. Any characters that are not representable in an XML document (such as those with ASCII values below 32) are transformed as [*nn*] where *nn* is the ASCII value of the character. • **filteredData** (string): The same print line as the rawData, except that any printer control characters have been filtered from the string. • **lineType** (string): The type of line, such as ItemSale. • **lineCategory** (string): The category of the line, such as Header or Trailer. • **font** (string): The font of the text. • **centered** (boolean): True if the line is centered. • **columnsToPrint** (integer): The number of printed columns the text, including blank spaces, consumes. • **linefeeds** (string): Indicates the number of lines advanced.
referenceNumber	String	The reference number of the tender. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.REFNUM$.
regEPrinted	Boolean	Indicates whether the Point of Sale Application printed a RegE receipt for an electronic tender. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.REGE.PRINTED.
responseCode	String	The response code id. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.RESPONSE.CODE$.
responseCodeDescriptor	String	.A description for the response code id. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable "JAVA.TENDER.RESPONSE.CODE.DESC$".
sequenceNumber	String	The sequence number of the tender. **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable JAVA.TENDER.RESPONSE.CODE.DESC$.
signatureNeeded	Boolean	Indicates whether the tender requires a signature to be obtained externally from the Point of Sale (typically on a physical paper receipt). **Note:** The 4690 Supermarket Application will provide this data if the payment system sets the global variable “JAVA.TENDER.SIGNATURE.NEEDED”.
tenderType	String	The type of tender (as defined by the POS application).
tenderVariety	String	The variety of tender (as defined by the POS application). **Note:** The concatenation of the tenderType and tenderVariety is used as a lookup key in tendermap.properties to determine the Java event to construct from the <tender> message.

TransactionStatus Events

transactionStart

Sent when a transaction is started within the POS application.

Note: Not sent for entry into training mode (GSA). Not sent for entry into transaction reentry.

XML Format

```
<transactionStart id="nnn" date="030828" time="15:23" type="cash"
  modifier="return" category="sales" accountNumber="zzzzzzz"
  returnPayments="true|false" cancelAllItems="true|false"/>
```

Table 63. <transactionStart> event

Value	Type	Description
accountNumber	String	Set to the layaway account number when starting a *layaway*, *layawayCancel*, or *layawayPayment* transaction type (GSA). Set to the charge plan account number when starting a *chargePlanA*, *chargePlanB*, *chargePlanC*, or *chargePlanD* transaction type (GSA).
cancelAllItems	Boolean	Included only when the transaction type is *layawayCancel*. Indicates whether the cancel applies to all items in the layaway account (GSA).
category	String	Either *sales* or *nonsales* depending on the transaction type.
date	String	The transaction's start date (YYMMDD).
id	String	A unique transaction identifier.
modifier	String	Additional transaction information that may be application specific. • return • send • vat • display
returnPayments	Boolean	Included only when the transaction type is *layawayCancel*. Indicates whether or not payments are returned to the customer (corresponds to customer-initiated) (GSA).
time	String	The transaction's start time (HH:MM).

Table 63. <transactionStart> event (continued)

Value	Type	Description
type	String	The transaction type. Expected values from GSA, SA, and ACE: • noSale • cash or regularSale that are equivalent • cashSpecial • cashDocument • cod • layaway • chargePlanA • chargePlanB • chargePlanC • chargePlanD • loan • pickup (also known as withdrawal) • tenderCount • totalsReadoutReset • itemMovementReport • itemPriceChange • setTransactionNumber • offlineReentry • training • voidPreviousTransaction • voidPreviousByLineItem • tenderListingReport • tenderRemoval • terminalMonitor • queryExchangeRate • tenderCashing • tenderExchange • priceVerify • terminalTransfer • terminalProgramLoad • itemReturn • wic • reprintPartialReceipt • reprintTenderReceipt • EBTBalanceInquiry • valueCardBalanceInquiry • departmentTotalsReport • layawayPayment • layawayCancel • priceChange • tenderFeeRefund • suspendedTransactionReport • modifyDepartmentPresets • openTransactionReport
trainingMode	String	Indicates whether or not the transaction started in training mode.
recoveryType	String	Indicates the recovery method when the transaction starts in recovery. Expected values from ACE: • recovery • retrieved • transfer
operatorID	String	The id of the currently logged in operator.

transactionVoid

Sent when a transaction is voided within the POS application.

Note: Not sent for exit from training mode, that is a signoff (GSA). Not sent for exit from transaction re-entry, that is a signoff.

XML Format

```
< transactionVoid id="nnn" type="xxx" category="sales|nonsales"
  date="yymmdd" time="hh:mm"/>
```

Table 64. <transactionVoid> event

Value	Type	Description
category	String	sales or nonsales (depending on the transaction type)
date	String	The date the transaction void occurred.
id	String	The identifier of the transaction being voided.
time	String	The time the transaction void occurred.

Table 64. <transactionVoid> event (continued)

Value	Type	Description
type	String	The transaction type. **Note:** It might not be possible to void all the transaction types listed.Expected values from GSA, SA, and ACE are: • noSale • cash or regularSale that is equivalent • cashSpecial • cashDocument • cod • layaway • chargePlanA • chargePlanB • chargePlanC • chargePlanD • loan • pickup (also known as withdrawal) • tenderCount • totalsReadoutReset • itemMovementReport • itemPriceChange • setTransactionNumber • offlineReentry • training • voidPreviousTransaction • voidPreviousByLineItem • tenderListingReport • tenderRemoval • terminalMonitor • queryExchangeRate • tenderCashing • tenderExchange • priceVerify • terminalTransfer • terminalProgramLoad • itemReturn • wic • reprintPartialReceipt • reprintTenderReceipt • EBTBalanceInquiry • valueCardBalanceInquiry • departmentTotalsReport • layawayPayment • layawayCancel • priceChange • tenderFeeRefund • suspendedTransactionReport • modifyDepartmentPresets • openTransactionReport

transactionSuspended

Sent whenever a transaction is suspended within the POS application.

XML Format

```
< transactionSuspended id="nnn" type="xxx" category="sales|nonsales"
  date="yymmdd" time="hh:mm"/>
```

Table 65. <transactionSuspended> event

Value	Type	Description
category	String	sales or nonsales (depending on the transaction type)
date	String	The date the suspend occurred.
id	String	The identifier of the transaction being suspended.
time	String	The time the suspend occurred.
type	String	The transaction type. Note: It might not be possible to suspend all the transaction type listed below. Expected values from GSA, SA, and ACE are: • noSale • cash or regularSale that is equivalent • cashSpecial • cashDocument • cod • layaway • chargePlanA • chargePlanB • chargePlanC • chargePlanD • loan • pickup (also known as withdrawal) • tenderCount • totalsReadoutReset • itemMovementReport • itemPriceChange • setTransactionNumber • offlineReentry • training • voidPreviousTransaction • voidPreviousByLineItem • tenderListingReport • tenderRemoval • terminalMonitor • queryExchangeRate • tenderCashing • tenderExchange • priceVerify • terminalTransfer • terminalProgramLoad • itemReturn“ • wic • reprintPartialReceipt • reprintTenderReceipt • EBTBalanceInquiry • valueCardBalanceInquiry • departmentTotalsReport • layawayPayment • layawayCancel • priceChange • tenderFeeRefund • suspendedTransactionReport • modifyDepartmentPresets • openTransactionReport

transactionEnd

Sent when a transaction is ended within the POS application.

XML Format

```
< transactionEnd id="nnn"" type="xxx" category="sales|nonsales" date="yymmdd" time="hh:mm"/>
```

Table 66. <transactionEnd> event

Value	Type	Description
category	String	sales or nonsales (depending on the transaction type)
date	String	The date the transaction ended.
id	String	The identifier of the transaction being ended.
time	String	The time the transaction ended.
type	String	The transaction type. Expected values from GSA, SA, and ACE are: • noSale • cash or regularSale that is equivalent • cashSpecial • cashDocument • cod • layaway • chargePlanA • chargePlanB • chargePlanC • chargePlanD • loan • pickup (also known as withdrawal) • tenderCount • totalsReadoutReset • itemMovementReport • itemPriceChange • setTransactionNumber • offlineReentry • training • voidPreviousTransaction • voidPreviousByLineItem • tenderListingReport • tenderRemoval • terminalMonitor • queryExchangeRate • tenderCashing • tenderExchange • priceVerify • terminalTransfer • terminalProgramLoad • itemReturn • wic • reprintPartialReceipt • reprintTenderReceipt • EBTBalanceInquiry • valueCardBalanceInquiry • departmentTotalsReport • layawayPayment • layawayCancel • priceChange • tenderFeeRefund • suspendedTransactionReport • modifyDepartmentPresets • openTransactionReport
total	String	The transaction total.
subtotal	String	The transaction subtotal.

Table 66. <transactionEnd> event (continued)

Value	Type	Description
tax	String	The amount of tax.
discountableSubTotal	String	The subtotal of all discountable items.

transactionUpdate

Sent when required to notify of a significant change in the transaction, depending on the event attributes. ended within the POS application.

XML Format

```
< transactionUpdate property_1="value_1" property_2="value_2"  />
```

Table 67. <transactionUpdate> event

Value	Type	Description
itemAllowanceAllowed	Boolean	Indicates when the item allowance function is valid in the transaction. (GSA)
itemDiscountAllowed	Boolean	Indicates when the item discount function is valid in the transaction (GSA, ACE)
itemVoidAllowed	Boolean	Indicates when the void line item function is valid in the transaction. (GSA, SA, ACE)
paymentSentForHostProcessing	Boolean	Indicates that an electronic payment was sent to the host for approval. (GSA, SA, ACE) For SA, the value of this attribute comes from the JAVA.PAYMENT.SENT.TO.HOST global variable which must be set by the payment system.
transactionDiscountAllowed	Boolean	Indicates when the transaction discount function is valid in the transaction. (GSA, SA, ACE)

transactionTaxChange

Sent:

- GSA: When a TAXCODE entry has been applied to the transaction. A <transactionTotals> event should follow this event.
- ACE & SA: When a transaction discount group has been applied that includes tax exemptions. This event should follow the <transactionDiscount> event, and be followed by a <transactionTotals> event.

XML Format

```
<transactionTaxChange taxType="taxExempt|alternateTax|manualTax"
  taxReason="cc" isVoided="true|false"/>
```

Table 68. <transactionTaxChange> event

Value	Type	Description
isVoided	Boolean	Indicates whether the tax change has been voided.
taxType	String	(GSA) Set to the type of taxcode. (SA) Set to *taxExempt* if the taxReason is a taxExempt discount (ACE) Set to *taxExempt* if the taxReason is a discount group that exempts all tax plans. Set to *alternateTax* if the taxReason is a discount group that exempts some tax plans.

Table 68. <transactionTaxChange> event (continued)

Value	Type	Description
taxReason	String	(GSA) The tax code. (SA, ACE) The discount group.

TransactionTotals event

transactionTotals

Sent when an application change occurs that causes an update to any of the transaction totals.

XML Format

```
<transactionTotals total="xxx.xx" tax="xxx.xx" subTotal="xxx.xx"
  balanceDue="xxx.xx" foodStampTotal="xxx.xx" foodStampBalance="xxx.xx"
  foodStampChange="xxx.xx" changeDue="xxx.xx  totalItems="n"
  totalSavings="xxx.xx" totalCouponCount="n" totalCouponAmount="xxx.xx"
  foreignBalanceDue="xxx.xx" tenderExchanged="xxx.xx"
  layawayFee="xx.xx" layawayDeposit="xx.xx" layawayBalanceDue="xxx.xx"/>
```

Note: When any of the transaction totals is updated, an event might be dispatched by the application to the framework containing just the values that have been updated.

Table 69. <transactionTotals> event

Value	Type	Description
balanceDue	String	The remaining tender that is necessary to complete the transaction.
changeDue	String	The cash amount due the customer.
foodStampBalance	String	The remaining transaction balance that might be satisfied using foodstamps. Should be included whenever a foodstamp total is taken. (SA, ACE).
foodStampChange	String	The foodstamp amount due the customer.
foodStampTotal	String	The total amount of all foodstamp appropriate items in the transaction. Should always be included (SA, ACE).
foreignBalanceDue	String	The remaining balance translated to the foreign currency as configured in the application. Should be included whenever a foreign total is taken. (SA, ACE, or GSA when multiple currency feature is present.)
layawayBalanceDue	String	For layaway transaction type, set to the transaction amount plus fee amount minus deposit amount (GSA). For layaway payment transaction type, set to the current layaway balance due (the prior layaway balance due minus payments) (GSA). For *layawayCancel* transaction type, set to the adjusted layaway balance for item cancels. Set to zero when the entire layaway account is cancelled. In this case, the amountDue indicates the amount of the refund due the customer (GSA).
layawayDeposit	String	For layaway transaction type, set to the layaway deposit amount (if any). For layawayCancel transaction type, use a negative layaway deposit if the deposit is refundable (GSA). For a *layawayPayment* transaction type, set to the layaway deposit amount (if any).

Table 69. <transactionTotals> event (continued)

Value	Type	Description
layawayFee	String	For layaway transaction type, set to the layaway fee (if any). For *layawayCancel* transaction type, use a negative layaway fee if the fee is refundable (GSA).
subTotal	String	The transaction total (excluding taxes).
tax	String	The transaction tax amount.
total	String	The transaction total amount (including taxes).
totalCouponAmount	String	The total value of coupons used in this transaction
totalCouponCount	String	The total number of coupons used in this transaction
totalItems	String	The total number of items in the transaction.
totalSavings	String	The amount saved by the loyalty customer for the transaction (as determined by the POS application).
tenderExchanged	String	The amount of tender accepted in this transaction.

WorkstationStatus event

workstationStatus

Sent to notify listeners that a change has occurred in the POS workstation (terminal). The terminalStatus property specifies the current state of the terminal. Additional workstation *properties* may also be specified if needed. WorkstationStatus events are typically used for events that are transitory (such as a *clear* or *acknowledge*), where a property value does not need to be maintained or persisted.

XML Format

```
<workstationStatus terminalStatus="xxxx"  additionalProperty1
  ="xxx" additionalProperty2 ="xxx" />
```

Table 70. <workstationStatus> event

Value	Type	Description
additionalPropertyN	String	Specify any other data needed for this event.
terminalStatus	String	The current status of the terminal device.

TransactionReplayComplete event

transactionReplayComplete

Sent to indicate that a transaction recovery, retrieve, or transfer has finished replaying all of its items.

XML Format

```
<transactionReplayComplete date = "100929" time = "16.01" transactionID = "23"/> >
```

Table 71. <TransactionReplayComplete> event

Value	Type	Description
date	String	The date (in YYMMYDD format) the transaction replay completed.
time	String	The time (in HH:MM format) the transaction replay completed.
transactionID	String	The id of the transaction.

Appendix G. SALogonActionImpl source example

This section explains how "action" classes work by walking through a particular class: SALogonActionImpl.java. This class is used to implement the logon function when any of the POSAutomationProvider.logon methods are called.

The POSAutomationProvider.logon method was either passed, or constructed an OperatorIdentifier and placed in the argument list of the ActionRequest. The OperatorIdentifier includes the operator ID and password. On line 64-66, the operator ID and passwords are retrieved and stored in member variables. If no operator id or password was provided on the logon call, defaults are retrieved from the applogon bundle on lines 70 and 74.

The first significant line of code in the performAction method is the call to the super.performAction method on line 105. The superclasses check for errors and allow the AEF Error Handler to attempt to clear them before proceeding with the action.

On line 129-132, the values of several SA states are retrieved into member variables.

On lines 144 and 151 a state check is done to see if SA is in the secure signoff state, or the signon state. If neither, an exception is thrown because SA is not in a proper state to perform the signon.

Assuming that SA in the signon state, line 168 calls a method to send the operator ID key sequence. If the operator ID key sequence is sent successfully, line 173 calls a method to send the operator password key sequence. If the operator password key sequence is sent successfully, line 178 calls a method to wait for the AEF to detect and create an Operator object to be returned.

Line 215 is the start of the sendIDSequence method. The SimpleKeySequenceActionImpl for sending the <operator-id><SLASH> key sequence is setup on lines 236-241. *args* is a HashMap that is used to contain the arguments for the key sequence action. The HashMap is cleared on line 236. On line 238, the sequence id is set. The sequence ID is used by the factory to lookup the corresponding key sequence in the ***sequence*** bundle (see “Key Sequence Bundle - error handling” on page 201). The key sequences defined in the bundle contain variables of the form *%n*. On line 237, the variable %O is given the value of the operator id.

The "good" conditions for this key sequence is defined on line 242. In this case, the single good condition is that SA ends up in the "password" state after sending the <operator-id><SLASH> key sequence. If there are multiple conditions defined in the condition array, they are logically or'd.

A new ConditionLock instance is created on line 248. This lock is used to block the calling thread while the key sequence is sent to the POS application and to wait for a good condition, bad condition, or a timeout to occur.

On line 253, the current instance number of the last Operator detected by the ObjectDetector is retrieved and stored. This is crucial to knowing when a new Operator instance has been created. This instance number is passed to an ObjectDetectorLock on line 435 to wait for a new instance of the Operator to be detected.

The next step is to have the ConditionLock perform the SimpleKeySequenceActionImpl to send the <operator-id><SLASH> key sequence as shown on line 254. The good conditions are passed, along with the standard set of bad conditions from SABadConditionsImpl.java. For SA, a common back condition is that the application goes into state 1 (clear state). The performActionAndWait method blocks until one of the good or bad conditions is observed, or a timeout occurs. The integer returned from the performActionAndWait method indicates which condition was observed. A zero return value indicates

the first good condition was observed. A *one* return value indicates the second good condition was observed, and so on. A *negative one* return value indicates the first bad condition was observed. A *negative two* indicates the second bad condition was observed, and so on. If a negative return value is observed, an AEFErrorHandler is created to attempt to deal with the error on lines 263 and 264.

Once the operator ID sequence has been sent successfully, the sendPasswordSequence method is called. This method that begins on line 297. This method is similar to the sendIDSequence method.

Once the operator password sequence has been sent successfully, the waitForOperator method is called. The waitForOperator method begins on line 420, and uses and ObjectDetectorLock to block the calling thread until a new Operator instance is detected, or until a timeout occurs. The Operator instance number that was saved earlier is passed to the ObjectDetectorLock on line 435 to ensure that only a new Operator is detected.

If a new Operator is successfully detected, the performAction return value is set to reference the new Operator instance.

```
1.  /*
2.  * SALogonActionImpl
3.  *
4.  * 07/29/2002
5.  *
6.  * Copyright:
7.  * Licensed Materials - Property of IBM
8.  * &cdqg;Restricted Materials of IBM&odq;
9.  * 5724-AEF
10. * (C) Copyright IBM Corp. 2004, 2007.
11. *
12. * fpz0xmp.ide, SATIN, rssv1r2 05/04/04
13. */
14. package com.ibm.retail.AEF.action;
15.
16. import com.ibm.retail.AEF.automation.*;
17. import com.ibm.retail.AEF.util.*;
18. import com.ibm.retail.si.util.*;
19. import com.ibm.retail.si.Copyright;
20. import com.ibm.retail.AEF.factory.*;
21. import com.ibm.retail.AEF.thread.*;
22. import com.ibm.retail.AEF.data.*;
23. import com.ibm.retail.AEF.session.*;
24.
25. import java.util.*;
26. import java.rmi.*;
27.
28. import org.apache.commons.logging.Log;
29. import org.apache.commons.logging.LogFactory;
30.
31. /**
32. * SALogonActionImpl is a class that the POSAutomationProvider
    uses to accomplish
33. * a logon onto the SA application.
34. *
35. */
36. public class SALogonActionImpl extends SAActionImpl
37. {
38.
39.    static String copyright()
40.         { return com.ibm.retail.si.Copyright.IBM_COPYRIGHT_SHORT; }
41.    private static AEFPerfTrace perfTrace = AEFPerfTrace.getInstance();
42.
43.    /**
44.     * Constructor
45.     *
46.     * @param request The ActionRequest that contains a HashMap of arguments.
47.     *
48.     * @exception AEFException
49.     */
50.    public SALogonActionImpl(ActionRequest request) throws AEFException
51.    {
52.       super(request);
53.
54.       if (log.isTraceEnabled())
55.       {
56.          tempAEFMessage.setMessage("+Enter ALogonActionImpl.SALogonActionImpl().");
57.          log.trace(tempAEFMessage);
58.       }
59.
60.       if (sessionID == null)
61.       {
62.          sessionID = "999";
63.       }
64.       OperatorIdentifier opId = (OperatorIdentifier)(request.getArguments());
65.       id = opId.getID();
66.       password = opId.getPassword();
67.       newPassword = opId.getNewPassword();
```

SALogonActionImpl source example

```
68.        if (id == null || id.length() == 0)
69.        {
70.           id = AppLogons.getID(sessionID);
71.        }
72.        if (password == null || password.length() == 0)
73.        {
74.           password = AppLogons.getPassword(sessionID);
75.        }
76.
77.        if (log.isTraceEnabled())
78.        {
79.           tempAEFMessage.setMessage("-Exit SALogonActionImpl.SALogonActionImpl
       ().");
80.           log.trace(tempAEFMessage);
81.        }
82.     }
83.
84.
85.     /**
86.      * Perform the action represented by the ActionRequest and return an
      ActionResult.
87.      *
88.      *
89.      * @param request The ActionRequest that contains the classname and
      arguments.
90.      * @return Object The Operator instance.
91.      * @exception AEFException
92.      * Among the possible error codes are:
93.      * <br>AEFConst.INVALID_ARGUMENT, AEFConst.NEW_PASSWORD_PROHIBITED
94.      * <br>AEFConst.APPLICATION_NOT_IN_PROPER_STATE
95.      */
96.     public Object performAction() throws AEFException
97.     {
98.        if (log.isTraceEnabled())
99.        {
100.          tempAEFMessage.setMessage("+Enter SALogonActionImpl.
101.                                    performAction().");
102.          log.trace(tempAEFMessage);
103.       }
104.
105.       super.performAction(); // Call super to perform any common
106.       // processing and clear any errors
107.       // before we start.
108.
109.       if (newPassword != null && newPassword.length() > 0)
110.       {
111.          String tempString =
112.                "SALogonActionImpl.performAction(): Change password
113.                function not implemented.";
114.          AEFException tempException = new AEFException(
115.                AEFConst.INVALID_ARGUMENT,
116.                AEFConst.NEW_PASSWORD_PROHIBITED,
117.                tempString);
118.          tempAEFMessage.setMessage(tempString);
119.          log.error(tempAEFMessage, tempException);
120.          throw tempException;
121.       }
122.
123.       Object retVal = null;
124.       int seqResult = -1;
125.       String currentState = getCurrentState();
126.       detector = ((DetectorAccess)(SessionContext.getSession())).
127.                       getOperatorDetector();
128.
129.       idState = State.getState("ID");
130.       passwordState = State.getState("PASSWORD");
131.       itemEntryState = State.getState("ITEMENTRY");
```

SALogonActionImpl source example

```
132.       securedSignoffState = State.getState("SECURED");
133.
134.       // Check if we are in a clear state.
135.       // This could be a B014 NO TRANSACTION RECOVERY
136.       // for example. If so, clear the error.
137.       if (currentState.equals(State.getState("CLEAR")))
138.       {
139.          sendClearSequence(true);
140.          currentState = getCurrentState();
141.       }
142.
143.
144.       if (currentState.equals(securedSignoffState))
145.       {
146.          // An operator is already signed on.
147.          retVal = doSecuredSignon();
148.       }
149.       //Check precondition that POS_STATE is &cdqg;ID&odq; state (2).
150.       //If not, throw exception.
151.       else if (!currentState.equals(idState))
152.       {
153.          // Not in valid state for signon.
154.
155.          String tempString = "SALogonActionImpl.performAction(): POS application "
                                 + "not in proper state to perform logon.\n Expected state "
                                 + idState + ", but application is in state " + currentState
                                 + ".";
156.          AEFException tempException = new AEFException(
157.                   AEFConst.APPLICATION_NOT_IN_PROPER_STATE,
158.                   0,
159.                   tempString);
160.                   tempAEFMessage.setMessage(tempString);
161.          log.error(tempAEFMessage, tempException);
162.          throw tempException;
163.       }
164.       else
165.       {
166.          // Application is in correct state to start preforming the logon.
167.
168.          seqResult = sendIDSequence();
169.          if (seqResult==0)
170.          {
171.             // Successfully got to password state,
172.             // send the password sequence.
173.             seqResult = sendPasswordSequence();
174.             if (seqResult == 0 ||
175.                 seqResult == 1)
176.             {
177.                Operator operator = null;
178.                operator = waitForOperator();
179.                retVal = operator;
180.                try
181.                {
182.                   operator.getInfo().setPassword(password);
183.                }
184.                catch (RemoteException re)
185.                {
186.                   // Can&csqg;t get here.
187.
188.                   tempAEFMessage.setMessage("There was a remote exception thrown in
                                                 SALogonActionImpl.performAction().");
189.                   log.error(tempAEFMessage, re);
190.                }
191.             }
192.          }
193.       }
194.
```

```
195.       if (log.isTraceEnabled())
196.       {
197.          tempAEFMessage.setMessage("-Exit SALogonActionImpl.performAction().");
198.          log.trace(tempAEFMessage);
199.       }
200.       return retVal;
201.    }
202.
203.
204.
205.    /**
206.     * Sends the key sequence for entering the operator id.
207.     *
208.     *
209.     * @return int 0 = success,
210.     * @exception AEFException
211.     * Among the possible error codes are:
212.     * <br>AEFConst.CONFIG_ERROR, AEFConst.NO_DEFAULT_ID_CONFIGURED
213.     */
214.
215.    public int sendIDSequence() throws AEFException
216.    {
217.       if (log.isTraceEnabled())
218.       {
219.          tempAEFMessage.setMessage("+Enter SALogonActionImpl.sendIDSequence().");
220.          log.trace(tempAEFMessage);
221.       }
222.       int retVal = -1;
223.
224.
225.       if ( id == null || id.length() == 0)
226.       {
227.          // Error, unable to lookup default id for terminal number.
228.
229.          String tempString = "SALogonActionImpl: Default ID and/or password
       not configured for terminal "
                               + sessionID + ", unable to perform logon.\n"
                               + "See applogon.properties for id/password configuration.";
230.          AEFException tempException = new AEFException(AEFConst.CONFIG_ERROR,
                                                 AEFConst.NO_DEFAULT_ID_CONFIGURED,
                                                 tempString);
231.          tempAEFMessage.setMessage(tempString);
232.          log.error(tempAEFMessage, tempException);
233.          throw tempException;
234.       }
235.
236.       args.clear();
237.       args.put("%O", id);
238.       args.put("SEQUENCE_ID", "id");
239.       keySequenceAction = (AEFAction)
240.             (actionFactory.makeAction(new
241.                ActionRequest("SimpleKeySequenceAction", args)));
242.       Condition[] goodConditions =
243.       {
244.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
245.                              POSDeviceProperties.POS_STATE,
246.                              passwordState), // 0 = success
247.       };
248.       lock = new ConditionLock();
249.
250.       // Save any current operator instance number so we
251.       // are sure to only get an operator
252.       // created after this key sequence is sent.
253.       instanceNumber = detector.getInstanceNumber();
```

```
254.       retVal = lock.performActionAndWait("wait-for-password-state",
255.                                          keySequenceAction,
256.                                          goodConditions,
257.                                          BadConditionsImpl.
258.                                            getInstance().getBadConditions(),
259.                                          getTimeout());
260.
261.       if (retVal < 0)
262.       {
263.          AEFErrorHandler errorHandler = new AEFErrorHandler("Send Signon ID
       Sequence Error");
264.          retVal = errorHandler.handleError(lock,
265.                                            keySequenceAction,
266.                                            goodConditions,
267.                                            BadConditionsImpl.
268.                                              getInstance().
269.                                              getBadConditions(),
270.                                            getTimeout(),
271.                                            retVal);
272.       }
273.       if (log.isDebugEnabled())
274.       {
275.          tempAEFMessage.setMessage("SALogonActionImpl.sendIDSequence()
       returning : " + retVal);
276.          log.debug(tempAEFMessage);
277.       }
278.       if (log.isTraceEnabled())
279.       {
280.          tempAEFMessage.setMessage("-Exit SALogonActionImpl.sendIDSequence().");
281.          log.trace(tempAEFMessage);
282.       }
283.       return retVal;
284.    }
285.
286.
287.
288.    /**
289.     * Sends the key sequence for entering the operator password.
290.     *
291.     *
292.     * @return int 0,1 = success
293.     * @exception AEFException
294.     * Among the possible error codes are:
295.     * <br>AEFConst.CONFIG_ERROR, AEFConst.NO_DEFAULT_ID_CONFIGURED
296.     */
297.    public int sendPasswordSequence() throws AEFException
298.    {
299.       if (log.isTraceEnabled())
300.       {
301.          tempAEFMessage.setMessage("+Enter SALogonActionImpl.sendPasswordSequence().");
302.          log.trace(tempAEFMessage);
303.       }
304.
305.       int retVal = -1;
306.
307.       if ( password == null || password.length() == 0)
```

```
308.       {
309.          // Error, unable to lookup default password for terminal number.
310.          AEFException ex = new AEFException(AEFConst.CONFIG_ERROR,
311.                                AEFConst.NO_DEFAULT_ID_CONFIGURED,
312.                                "SALogonActionImpl: Default ID and/or password not "
                                   + "configured for terminal " + sessionID + ", unable to "
                                   + "perform logon.\nSee applogon.properties for "
                                   + "id/password configuration.");
313.          try
314.          {
315.             sendClearSequence();
316.          }
317.          catch (AEFException e)
318.          {
319.             // Ignore any errors from sending the clear.
320.          }
321.          tempAEFMessage.setMessage("SALogonActionImpl: Default ID and/or password not "
                                        + "configured for terminal " + sessionID + ", unable "
                                        + "to perform logon.\nSee applogon.properties for "
                                        + "id/password configuration.");
322.          log.error(tempAEFMessage, ex);
323.          throw ex;
324.       }
325.
326.
327.       // We have a password that we can use to attempt the
328.       // secure mode signon.
329.       args.clear();
330.       args.put("%0", password);
331.       args.put("SEQUENCE_ID", "password");
332.       keySequenceAction = (AEFAction)
333.       (actionFactory.makeAction(new ActionRequest("SimpleKeySequenceAction",
334.                                                   args)));
335.
336.       Condition andConds1[] =
337.       {
338.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
339.                POSDeviceProperties.POS_STATE,
340.                State.getState("MAIN")),
341.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
342.                POSDeviceProperties.POS_SUB_STATE,
343.                Substate.getSubstate("SELECT_PROCEDURE"))
344.
345.       };
346.
347.       Condition andConds2[] =
348.       {
349.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
350.                POSDeviceProperties.POS_STATE,
351.                State.getState("MAIN")),
352.          new PropertyEqualsCondition(POSDeviceProperties.CATEGORY,
353.                POSDeviceProperties.POS_SUB_STATE,
354.                Substate.getSubstate("CHANGE_OR_BAL_DUE"))
355.
356.       };
357.
358.       Condition goodConditions[] =
359.       {
360.          new AndCondition(andConds1),
361.          new AndCondition(andConds2)
362.       };
363.
364.       if (lock == null)
365.       {
366.          lock = new ConditionLock();
```

```
367.       }
368.       if (log.isDebugEnabled())
369.       {
370.          tempAEFMessage.setMessage("About to wait on the perform action
       for &csqg;password&csqg;");
371.          log.debug(tempAEFMessage);
372.       }
373.       if (perfTrace.isEnabled(AEFPerfTrace. COARSE))
374.       {
375.          perfTrace.reportTimer(AEFPerfTrace. COARSE,
376.                      sessionID,
377.                      "logon",
378.                      ">>>Sending logon password key sequence to application.");
379.       }
380.       retVal= lock.performActionAndWait("wait-for-item-entry-state",
381.                                         keySequenceAction,
382.                                         goodConditions,
383.                                         BadConditionsImpl.
384.                                           getInstance().getBadConditions(),
385.                                         getTimeout());
386.       if (retVal < 0)
387.       {
388.          AEFErrorHandler errorHandler = new AEFErrorHandler
       ("Send Signon Password Sequence Error");
389.          retVal = errorHandler.handleError(lock,
390.                                            keySequenceAction,
391.                                            goodConditions,
392.                                            BadConditionsImpl.
393.                                              getInstance().
394.                                              getBadConditions(),
395.                                            getTimeout(),
396.                                            retVal);
397.       }
398.       if (log.isDebugEnabled())
399.       {
400.          tempAEFMessage.setMessage(" Condition returned was " + retVal);
401.          log.debug(tempAEFMessage);
402.       }
403.
404.       if (log.isTraceEnabled())
405.       {
406.          tempAEFMessage.setMessage("-Exit SALogonActionImpl.sendPasswordSequence().");
407.          log.trace(tempAEFMessage);
408.       }
409.       return retVal;
410.    }
411.
412.
413.
414.    /**
415.     * Blocks the calling thread until a new operator object is created.
416.     *
417.     *
418.     * @exception AEFException
419.     */
420.    public Operator waitForOperator() throws AEFException
421.    {
422.       if (log.isTraceEnabled())
423.       {
424.          tempAEFMessage.setMessage("+Enter SALogonActionImpl.waitForOperator().");
425.          log.trace(tempAEFMessage);
```

```
426.       }
427.
428.       Operator retVal = null;
429.       // Successfully got to item entry. Wait for the operator
430.       // object to be created.
431.       ObjectDetectorLock objLock = new ObjectDetectorLock();
432.       retVal = (Operator)
433.          (objLock.waitForNewObject("wait-for-operator",
434.                                    detector,
435.                                    instanceNumber,
436.                                    getTimeout()));
437.       if (perfTrace.isEnabled(AEFPerfTrace. COARSE))
438.       {
439.          perfTrace.reportTimer(AEFPerfTrace. COARSE,
440.                                sessionID,
441.                                "logon",
442.                                "<<<Detected operator logon from application.");
443.       }
444.
445.       if (log.isTraceEnabled())
446.       {
447.          tempAEFMessage.setMessage("-Exit SALogonActionImpl.waitForOperator().");
448.          log.trace(tempAEFMessage);
449.       }
450.       return retVal;
451.
452.    }
453.
454.
455.    /**
456.     * Performs the logic required to sign an operator on from secured mode.
457.     *
458.     *
459.     * @exception AEFException
460.     * Among the possible error codes are:
461.     * <br>AEFConst.PROCEDURE_NOT_ALLOWED,
462.     * AEFConst.ANOTHER_OPERATOR_SIGNED_ON
463.     */
464.    public Operator doSecuredSignon() throws AEFException
465.    {
466.       if (log.isTraceEnabled())
467.       {
468.          tempAEFMessage.setMessage("+Enter SALogonActionImpl.doSecuredSignon().");
469.          log.trace(tempAEFMessage);
470.       }
471.       if (log.isDebugEnabled())
472.       {
473.          tempAEFMessage.setMessage("App in secured signoff state.");
474.          log.debug("App in secured signoff state.");
475.       }
476.       Operator operator = null;
477.       // Somebody is already signed on, check to see
478.       // if the id of the new signon request
479.       // matches the id of the already signed on operator.
480.
```

```
481.       try
482.       {
483.          operator = automationProvider.getOperator();
484.          if (operator!=null && !operator.getInfo().getID().equals(id))
485.          {
486.             // Currently signed on operator is a different operator.
487.
488.             String tempString = "&cdqg;SALogonActionImpl.performAction():
         Can&csqg;t perform "
                                  + "logon for operator " + id + " because operator "
                                  + operator.getInfo().getID()+ " is already signed on.";
489.             AEFException tempException = new
490.                   AEFException(AEFConst.PROCEDURE_NOT_ALLOWED,
491.                                AEFConst.ANOTHER_OPERATOR_SIGNED_ON,
492.                                tempString);
493.             tempAEFMessage.setMessage(tempString);
494.             log.error(tempAEFMessage, tempException);
495.             throw tempException;
496.          }
497.          else if (operator != null)
498.          {
499.             // The requested operator is already
500.             // signed on, but app is in secured
501.             // signoff mode, just need to send password sequence.
502.
503.             // Use the password of the currently
504.             // signed on operator if possible.
505.             String pwd = operator.getInfo().getPassword();
506.             if (pwd != null && pwd.length() > 0)
507.             {
508.                password = pwd;
509.             }
510.             int seqResult = sendPasswordSequence();
511.             if (seqResult == 0 ||
512.                 seqResult == 1)
513.             {
514.                operator = waitForOperator();
515.                try
516.                {
517.                   operator.getInfo().setPassword(password);
518.                }
519.                catch (RemoteException re)
520.                {
521.                   // Can&csqg;t get here.
522.
523.                tempAEFMessage.setMessage("There was a remote exception thrown in
                                             SALogonActionImpl.doSecuredSignon().");
524.                log.error(tempAEFMessage, re);
525.                }
526.             }
527.          }
528.       }
```

```
529.       catch (RemoteException re)
530.       {
531.          // Not expected to get here since we are calling locally.
532.
533.          tempAEFMessage.setMessage("There was a remote exception thrown in
                                        SALogonActionImpl.doSecuredSignon().");
534.          log.error(tempAEFMessage, re);
535.       }
536.
537.       if (log.isTraceEnabled())
538.       {
539.          tempAEFMessage.setMessage("-Exit SALogonActionImpl.doSecuredSignon().");
540.          log.trace(tempAEFMessage);
541.       }
542.       return operator;
543.    }
544.
545.    /* Instance Variables */
546.    protected AEFAction keySequenceAction;
547.    protected String id;
548.    protected String password;
549.    protected String newPassword;
550.    protected ConditionLock lock;
551.    protected ObjectDetector detector;
552.    protected int instanceNumber;
553.    protected String idState;
554.    protected String passwordState;
555.    protected String itemEntryState;
556.    protected String securedSignoffState;
557.
558.    private static Log log = LogFactory.getLog(SALogonActionImpl.class);
```

Appendix H. POS device support

POS device support

Access to POS devices by the underlying POS application (ACE, SA, GSA) requires sending binary commands to the POS device through 4690 OS calls. SI intercepts the application calls to the POS device and translates these binary commands into JavaPOS statements. JavaPOS is intended to be device agnostic, so some commands differ in implementation from JavaPOS verses 4690 commands. Store Integrator doesn't currently support all the functions available for each POS device.

POS printer

When you print using the native applications, ACE, SA, and GSA, binary commands are sent to the 4610 printer via 4690 calls.

SI intercepts the application's calls to the 4610 printer and translates the binary commands into JavaPOS statements. However, not all commands are processed by SI and will not produce the expected results on the printer receipt. Some commands are processed and discarded, while others are partially interpreted and sent to JavaPOS.

Table 70 describes the known commands and the level of implementation completed by SI. At this time, SI does not support font downloads. Some codes might behave differently on a native printer versus a redirected printer due to the implementation of JavaPOS.

Table 72. Printer codes supported by Store Integrator

Command	Native Code	OK	Special Note
Normal character set		Yes	
Horizontal Tab with reservations - no dot positions	0x09	Partial	Not to dot positions. Tabs to approximate space.
Line feed and format commands	0x0a, 0x0d, 0c0c	Yes	
Set Print Mode	1B 21 n	No	Support implemented for bold and Double. Reverse, Video, Italics and Overline not supported. Underline Not supported
Set Cancel Double Wide	1B 57 n	Yes	
Set Cancel Underline	1B 2D n	Yes	
Set Cancel Invert	1B 48 n	Yes	
Set Cancel Emphasized Printing	1B 47 n	Yes	
Set Cancel High Quality Printing	1B 2F n	Yes	
Set Print Station	1B 63 30 n	Yes	
Set User Defined / Resident Chars	1B 25 n	No	

Table 72. Printer codes supported by Store Integrator (continued)

Command	Native Code	OK	Special Note
Set Code Page	1B 74 n	No	
Set Inter-Character Spacing	1B 20 n	No	
Set Cancel Rotate Characters	1B 56 n	Yes	
Enable Disable upside down print	1B 7B n	No	
Select Print Station for Station	1B 63 31 n	Yes	
Select 1/8 inch spacing	1B 31	Yes	
Select 1/6 inch spacing	1B 32	Yes	
Set Line Spacing using Minimum Unit	1B 33 n	Yes	Spacing unit differences in JavaPOS
Set Sheet Eject Length	1B 43 n	Yes	
Set Horizontal Tab Positions	1B 44 n1 n2...00 00	Yes	Code accepts spaces, but JavaPOS doesn't go to DOTS per native
Set Left Margin	1B 24 n1 n2	No	
Set Relative Position	1B 5C n1 n2	No	
Align Positions	1B 61 n	Yes	
Set Cancel Unidirectional Printing	1B 55 n	No	
Set Error Recovery	1B 63 34 n	No	
Define Document Wait time	1B 66 x y	Yes	
Character for Reprinted Lines	1B 2B n	Yes	
Print Bar Code	1D 6B n data 00	Yes	
Select Horizontal Size of Barcode	1D 77 n	No	
Select Vertical Size of BarCode	1D 68 n	No	
Select Printing Position of Bar Code	1D 48 n	Yes	
Select font for HRI	1D 66 n	No	
Print PDF417 Bar Code	1D 50 data 00	Yes	
Print PDF417 ECC	1D 52 n1 n2	No	
Select Aspect Ratio PDF417 Bar Code	1D 53 r c	No	
Enable PDF417 Truncation	1D 54 n	No	
Print Feed Paper using minimum units	1B 4A n	Yes	Feed NUMBER NOT SAME IN JavaPOS VS NATIVE

Table 72. Printer codes supported by Store Integrator (continued)

Command	Native Code	OK	Special Note
Select and Print a graphics Logo	1B 2A d w h data	Yes	
Print Predefined Graphics Logo	1D 2F m logo#	Yes	
Print Predefined Messages	1D 5E message #	Yes	
Proportional Fonts	1B 26 s n m data	No	
Fix Font Matrix	1B 3A n	Yes	
InLine Logo Printing	1D 4A d n1 n2 data	No	
Select character set Scalable font	1D 21 n	No	
Select Thermal Paper	1D 3B n	No	
Select Color Printing	1B 72 n	No	
Enable Disable Beeper	1B 07 n n2	No	
Continuation of previous command	1B 2E	Yes	
Return Home Select Print head	1B 3C n	Yes	
Paper Cut / DI Eject	1B 6D	Yes	
Retrieve the User Flash Storage	04 1B 34 n1 n2 a1 a2 a3	Yes	
Flip Check	1B 35	No	
Start Document Scan	1B 3E n	No	(TI 8/9)
Print Scanned Image	1B 30 n1h.....	No	(TI 8/9)
Store Scanned Image	1B 41 n1h.....	No	(TI 8/9)
Retrieve Scanned Image	1B 39 ih.....	No	(TI 8/9)
Scanner Calibration	1D 63 30	No	
Retrieve Next Image Location	1D 4E 01 00	No	(TI 8/9)
Retrieve First Unread Image Location	1d 4E 01 01	No	(TI 8/9)
Set Compression Format	1B 4D 1F n1 n2	No	(TI 8/9)
Hold Buffer	1B 37	Yes	
Release Buffer	10 05 31	Yes	
MicroTolerance - Request	04 1B 53 n	Yes	
MicroTolerance - Loading	04 1B 4D n h l	Yes	
Retrieve User Flash Memory	04 1B 34 a1 a2 a3	Yes	

Table 72. Printer codes supported by Store Integrator (continued)

Command	Native Code	OK	Special Note
MICR Read	04 1B 49	No	
Status Sent to the System	04 1B 29 n	Yes	
User Flash Storage Size Request	1B 34 08 FF FF FF	Yes	
Erase Flash EPROM Sector	04 1B 23 n	Yes	
Download Graphics Logos	1D 2A logo# n1 n2 data	Yes	
Store Predefined Messages	1D 3A message# data 1D 3A	Yes	
User Defined Character	1B 26 s n m data	Yes	
Write to user Flash Storage	1B 27 n1 a1 a2 a3 data	Yes	
Erase Flash EPROM sector	04 1B 23 n	Yes	
Send checksum of Flash EPROM sector	1B 22 n	Yes	
Microcode Tolerance Information	04 1B 4D n h l	Yes	
Microcode Tolerance Information	04 1B 53 n	Yes	

Notices

This information was developed for products and services offered in the U.S.A.

Toshiba Global Commerce Solutions may not offer the products, services, or features discussed in this document in other countries. Consult your local Toshiba Global Commerce Solutions representative for information on the products and services currently available in your area. Any reference to a Toshiba Global Commerce Solutions product, program, or service is not intended to state or imply that only that Toshiba Global Commerce Solutions product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Toshiba Global Commerce Solutions intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-Toshiba Global Commerce Solutions product, program, or service.

Toshiba Global Commerce Solutions may have patents or pending patent applications covering the subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

Toshiba Global Commerce Solutions
Attn: General Counsel
3039 E. Cornwallis Rd.
RTP, NC 27709
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: TOSHIBA GLOBAL COMMERCE SOLUTIONS PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. Toshiba Global Commerce Solutions may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time without notice.

Toshiba Global Commerce Solutions may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Any references in this information to non-Toshiba Global Commerce Solutions Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this Toshiba Global Commerce Solutions product and use of those Web sites is at your own risk.

Information concerning non-Toshiba Global Commerce Solutions products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Toshiba Global Commerce Solutions has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Toshiba Global Commerce Solutions products. Questions on the capabilities of non-Toshiba Global Commerce Solutions products should be addressed to the suppliers of those products.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

Telecommunication regulatory statement

This product is not intended to be connected directly or indirectly by any means whatsoever to interfaces of public telecommunications networks, nor is it intended to be used in a public services network.

Electronic emission notices

When you attach a monitor to the equipment, you must use the designated monitor cable and any interference suppression devices that are supplied with the monitor.

Federal Communications Commission statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case the user will be required to correct the interference at his own expense.

Properly shielded and grounded cables and connectors must be used in order to meet FCC emission limits. Toshiba Global Commerce Solutions is not responsible for any radio or television interference caused by using other than recommended cables and connectors or by unauthorized changes or modifications to this equipment. Unauthorized changes or modifications could void the user's authority to operate the equipment.

This device complies with part 15 of the FCC Rules. Operation is subject to the following two conditions:

1. This device may not cause harmful interference, and
2. This device must accept any interference received, including interference that may cause undesired operation.

Industry Canada Class A Emission Compliance statement

This Class A digital apparatus complies with Canadian ICES-003.

Avis de conformité à la réglementation d'Industrie Canada

Cet appareil numérique de la classe A est conforme à la norme NMB-003 du Canada.

European Union Electromagnetic Compatibility (EMC) Directive Conformance Statement

This product is in conformity with the protection requirements of EU Council Directive 2004/108/EC on the approximation of the laws of the Member States relating to electromagnetic compatibility. Toshiba Global Commerce Solutions cannot accept responsibility for any failure to satisfy the protection requirements resulting from a non-recommended modification of the product, including the fitting of non-Toshiba Global Commerce Solutions option cards.

This product has been tested and found to comply with the limits for Class A Information Technology Equipment according to CISPR 22/European Standard EN 55022. The limits for Class A equipment were derived for commercial and industrial environments to provide reasonable protection against interference with licensed communication equipment.

Attention: This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

Responsible manufacturer:

Toshiba Global Commerce Solutions
3039 Cornwallis Road
Building 307
Research Triangle Park, North Carolina 27709
United States of America

European Community contact:

Toshiba Global Commerce Solutions
Brand Manager - Europe, Middle East & Africa
3 NEW SQUARE, FELTHAM, TW14 8HB Great Britain
Building: | Floor: NA | Office: MOBILE
Tel: 44-7967-275819
e-mail: robin_lyon@uk.ibm.com

European Community (EC) Mark of Conformity Statement

This product is in conformity with the protection requirements of EC Council Directive 89/336/EEC on the approximation of the laws of the Member States relating to electromagnetic compatibility. Toshiba Global Commerce Solutions cannot accept responsibility for any failure to satisfy the protection requirements resulting from a non-recommended modification of the product, including the fitting of non-Toshiba Global Commerce Solutions option cards.

This product has been tested and found to comply with the limits for Class A Information Technology Equipment according to CISPR 22/European Standard EN 55022. The limits for Class A equipment were derived for commercial and industrial environments to provide reasonable protection against interference with licensed communication equipment.

Warning: This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

Germany Class A statement

Deutschsprachiger EU Hinweis: Hinweis für Geräte der Klasse A EU-Richtlinie zur Elektromagnetischen Verträglichkeit

Dieses Produkt entspricht den Schutzanforderungen der EU-Richtlinie 2004/108/EG zur Angleichung der Rechtsvorschriften über die elektromagnetische Verträglichkeit in den EU-Mitgliedsstaaten und hält die Grenzwerte der EN 55022 Klasse A ein.

Um dieses sicherzustellen, sind die Geräte wie in den Handbüchern beschrieben zu installieren und zu betreiben. Des Weiteren dürfen auch nur von der Toshiba Global Commerce Solutions empfohlene Kabel angeschlossen werden. Toshiba Global Commerce Solutions übernimmt keine Verantwortung für die Einhaltung der Schutzanforderungen, wenn das Produkt ohne Zustimmung der Toshiba Global Commerce Solutions verändert bzw. wenn Erweiterungskomponenten von Fremdherstellern ohne Empfehlung der Toshiba Global Commerce Solutions gesteckt/eingebaut werden.

EN 55022 Klasse A Geräte müssen mit folgendem Warnhinweis versehen werden: “Warnung: Dieses ist eine Einrichtung der Klasse A. Diese Einrichtung kann im Wohnbereich Funk-Störungen verursachen; in diesem Fall kann vom Betreiber verlangt werden, angemessene Maβnahmen zu ergreifen und dafür aufzukommen.”

Deutschland: Einhaltung des Gesetzes über die elektromagnetische Verträglichkeit von Geräten

Dieses Produkt entspricht dem “Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG)”. Dies ist die Umsetzung der EU-Richtlinie 2004/108/EG in der Bundesrepublik Deutschland.

Zulassungsbescheinigung laut dem Deutschen Gesetz über die elektromagnetische Verträglichkeit von Geräten (EMVG) (bzw. der EMC EG Richtlinie 2004/108/EG) für Geräte der Klasse A

Dieses Gerät ist berechtigt, in Übereinstimmung mit dem Deutschen EMVG das EG-Konformitätszeichen - CE - zu führen.

Verantwortlich für die Einhaltung der EMV Vorschriften ist der Hersteller:

Toshiba Global Commerce Solutions
3039 Cornwallis Road
Building 307
Research Triangle Park, North Carolina 27709
United States of America

Der verantwortliche Ansprechpartner des Herstellers in der EU ist:

Toshiba Global Commerce Solutions
Brand Manager - Europe, Middle East & Africa
3 NEW SQUARE, FELTHAM, TW14 8HB Great Britain
Building: | Floor: NA | Office: MOBILE
Tel: 44-7967-275819
e-mail: robin_lyon@uk.ibm.com

Generelle Informationen:

Das Gerät erfüllt die Schutzanforderungen nach EN 55024 und EN 55022 Klasse A.

Australia and New Zealand Class A statement

Attention: This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

People's Republic of China Class A electronic emission statement

Attention: This is a Class A product. In a domestic environment this product may cause radio interference, in which case the user may be required to take adequate measures.

中华人民共和国“A 类”警告声明

声 明

此为 A 级产品，在生活环境中，该产品可能会造成无线电干扰。在这种情况下，可能需要用户对其干扰采取切实可行的措施。

Russian Electromagnetic Interference (EMI) Class A statement

ВНИМАНИЕ! Настоящее изделие относится к классу А.
В жилых помещениях оно может создавать радиопомехи,
снижения которых необходимы дополнительные меры

Japanese Electrical Appliance and Material Safety Law statement

本製品およびオプションに電源コードセットが付属する場合は、それぞれ
その装置専用のものになっていますので他の機器には使用しないで下さい。

Japanese power line harmonics compliance statement

高調波ガイドライン適合品

高調波ガイドライン適合品

Japan Voluntary Control Council for Interference Class A statement

Attention: This is a Class A product based on the standard of the Voluntary Control Council for Interference (VCCI). If this equipment is used in a domestic environment, radio interference may occur, in which case, the user may be required to take corrective actions.

この装置は、クラスA情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起こすことがあります。この場合には使用者が適切な対策を講ずるよう要求されることがあります。 VCCI－A

Japan Electronics and Information Technology Industries Association (JEITA) statement

高調波ガイドライン準用品

Japan Electronics and Information Technology Industries Association (JEITA) Confirmed Harmonics Guidelines with Modifications (products greater than 20 A per phase).

Korean communications statement

Please note that this device has been approved for business purposes with regard to electromagnetic interference (Type A). If you find this is not suitable for your use, you may exchange it for a non-business purpose one.

이 기기는 업무용(A급)으로 전자파적합기기로서 판매자 또는 사용자는 이 점을 주의하시기 바라며, 가정외의 지역에서 사용하는 것을 목적으로 합니다.

Taiwanese Class A compliance statement

警告使用者:
這是甲類的資訊產品，在居住的環境中使用時，可能會造成射頻干擾，在這種情況下，使用者會被要求採取某些適當的對策。

Cable ferrite requirement

All cable ferrites are required to suppress radiated EMI emissions and must not be removed.

Electrostatic discharge (ESD)

Attention: Electrostatic discharge (ESD) damage can occur when there is a difference in charge between the part, the product, and the service person. No damage will occur if the service person and the part being installed are at the same charge level.

ESD damage prevention

Anytime a service action involves physical contact with logic cards, modules, back-panel pins, or other ESD sensitive (ESDS) parts, the service person must be connected to an ESD common ground point on the product through the ESD wrist strap and cord.

The ESD ground clip can be attached to any frame ground, ground braid, green wire ground, or the round ground prong on the AC power plug. Coax or connector outside shells can also be used.

Handling removed cards

Logic cards removed from a product should be placed in ESD protective containers. No other object should be allowed inside the ESD container with the logic card. Attach tags or reports that must accompany the card to the outside of the container.

Product recycling and disposal

This unit must be recycled or discarded according to applicable local and national regulations. Toshiba Global Commerce Solutions encourages owners of information technology (IT) equipment to responsibly recycle their equipment when it is no longer needed. Toshiba Global Commerce Solutions offers a variety of product return programs and services in several countries to assist equipment owners in recycling their IT products. Information on Toshiba Global Commerce Solutions product recycling offerings can be found on the Toshiba Global Commerce Solutions product recycling programs website.

Español: Esta unidad debe reciclarse o desecharse de acuerdo con lo establecido en la normativa nacional o local aplicable. Toshiba Global Commerce Solutions recomienda a los propietarios de equipos de tecnología de la información (TI) que reciclen responsablemente sus equipos cuando éstos ya no les sean útiles. Toshiba Global Commerce Solutions dispone de una serie de programas y servicios de devolución de productos en varios países, a fín de ayudar a los propietarios de equipos a reciclar sus productos de TI. Se puede encontrar información sobre las ofertas de reciclado de productos de Toshiba Global Commerce Solutions en el sitio web Toshiba Global Commerce Solutions product recycling programs.

Notice: This mark applies only to countries within the European Union (EU) and Norway.

Appliances are labeled in accordance with European Directive 2002/96/EC concerning waste electrical and electronic equipment (WEEE). The Directive determines the framework for the return and recycling of used appliances as applicable throughout the European Union. This label is applied to various products to indicate that the product is not to be thrown away, but rather reclaimed upon end of life per this Directive.

Remarque : Cette marque s'applique uniquement aux pays de l'Union Européenne et à la Norvège. L'etiquette du système respecte la Directive européenne 2002/96/EC en matière de Déchets des Equipements Electriques et Electroniques (DEEE), qui détermine les dispositions de retour et de recyclage applicables aux systèmes utilisés à travers l'Union européenne. Conformément à la directive, ladite étiquette précise que le produit sur lequel elle est apposée ne doit pas être jeté mais être récupéré en fin de vie.

注意: このマークは EU 諸国およびノルウェーにおいてのみ適用されます。

この機器には、EU 諸国に対する廃電気電子機器指令 2002/96/EC(WEEE) のラベルが貼られています。この指令は、EU 諸国に適用する使用済み機器の回収とリサイクルの骨子を定めています。このラベルは、使用済みになった時に指令に従って適正な処理をする必要があることを知らせるために種々の製品に貼られています。

In accordance with the European WEEE Directive, electrical and electronic equipment (EEE) is to be collected separately and to be reused, recycled, or recovered at end of life. Users of EEE with the WEEE marking per Annex IV of the WEEE Directive, as shown above, must not dispose of end of life EEE as unsorted municipal waste, but use the collection framework available to customers for the return, recycling, and recovery of WEEE. Customer participation is important to minimize any potential effects of EEE on the environment and human health due to the potential presence of hazardous substances in EEE. For proper collection and treatment, contact your local Toshiba Global Commerce Solutions representative.

Disposal of IT products should be in accordance with local ordinances and regulations.

Battery return program

This product may contain sealed lead acid, nickel cadmium, nickel metal hydride, lithium, or lithium ion battery. Consult your user manual or service manual for specific battery information. The battery must be recycled or disposed of properly. Recycling facilities may not be available in your area. For information on disposal of batteries outside the United States, go to the Battery disposal website or contact your local waste disposal facility.

For Taiwan:

Please recycle batteries.

Taiwan contact information

台灣TGCS產品服務聯絡方式：
台灣東芝全球商業解決方案有限公司
台北市松仁路7號3樓
電話：0800-016-888

Toshiba Global Commerce Solution Product Service Contact Info:
Toshiba Global Commerce Solutions, Inc.
3F, No 7, Song Ren Road, Taipei Taiwan

Toshiba Global Commerce Solutions Taiwan Product Service Contact Info:
Toshiba Global Commerce Solutions Taiwan Corporation
3F, No 7, Song Ren Road, Taipei Taiwan
Telephone: 0800-016-888

For the European Union:

Notice: This mark applies only to countries within the European Union (EU)

Batteries or packaging for batteries are labeled in accordance with European Directive 2006/66/EC concerning batteries and accumulators and waste batteries and accumulators. The Directive determines the framework for the return and recycling of used batteries and accumulators as applicable throughout the European Union. This label is applied to various batteries to indicate that the battery is not to be thrown away, but rather reclaimed upon end of life per this Directive.

Les batteries ou emballages pour batteries sont étiquetés conformément aux directives européennes 2006/66/EC, norme relative aux batteries et accumulateurs en usage et aux batteries et accumulateurs usés. Les directives déterminent la marche à suivre en vigueur dans l'Union Européenne pour le retour et le recyclage des batteries et accumulateurs usés. Cette étiquette est appliquée sur diverses batteries pour indiquer que la batterie ne doit pas être mise au rebut mais plutôt récupérée en fin de cycle de vie selon cette norme.

バッテリーあるいはバッテリー用のパッケージには、EU 諸国に対する廃電気電子機器指令 2006/66/EC のラベルが貼られています。この指令は、バッテリーと蓄電池、および廃棄バッテリーと蓄電池に関するものです。この指令は、使用済みバッテリーと蓄電池の回収とリサイクルの骨子を定めているもので、EU 諸国にわたって適用されます。このラベルは、使用済みになったときに指令に従って適正な処理をする必要があることを知らせるために種々のバッテリーに貼られています。

In accordance with the European Directive 2006/66/EC, batteries and accumulators are labeled to indicate that they are to be collected separately and recycled at end of life. The label on the battery may also include a chemical symbol for the metal concerned in the battery (Pb for lead, Hg for mercury and Cd for cadmium). Users of batteries and accumulators must not dispose of batteries and accumulators as unsorted municipal waste, but use the collection framework available to customers for the return, recycling and treatment of batteries and accumulators. Customer participation is important to minimize any potential effects of batteries and accumulators on the environment and human health due to the potential presence of hazardous substances. For proper collection and treatment, contact your local Toshiba Global Commerce Solutions representative.

This notice is provided in accordance with Royal Decree 106/2008 of Spain: The retail price of batteries, accumulators and power cells includes the cost of the environmental management of their waste.

For California:

Perchlorate material – special handling may apply

Refer to www.dtsc.ca.gov/hazardouswaste/perchlorate.

The foregoing notice is provided in accordance with *California Code of Regulations Title 22, Division 4.5, Chapter 33: Best Management Practices for Perchlorate Materials*. This product/part includes a lithium manganese dioxide battery which contains a perchlorate substance.

Flat panel displays

The fluorescent lamp in the liquid crystal display contains mercury. Dispose of it as required by local ordinances and regulations.

Monitors and workstations

Connecticut: Visit the website of the Department of Environmental Protection at www.st.gov/dev for information about recycling covered electronic devices in the State of Connecticut, or telephone the Connecticut Department of Environmental Protection at 1-860-424-3000.

Oregon: For information regarding recycling covered electronic devices in the state of Oregon, go to the Oregon Department of Environmental Quality site at www.deq.state.or.us/kj/electronics.htm.

Washington: For information about recycling covered electronic devices in the State of Washington, go to the Department of Ecology Website at fortress.wa.gov/ecy/recycle/ or telephone the Washington Department of Ecology at 1-800-Recycle.

Trademarks

The following are trademarks or registered trademarks of Toshiba, Inc. in the United States or other countries, or both:

Toshiba
The Toshiba logo

The following are trademarks of Toshiba Global Commerce Solutions in the United States or other countries, or both:

AnyPlace
SureMark
SurePoint
SurePOS
TCxWave

The following are trademarks of International Business Machines Corporation in the United States or other countries, or both:

DB2
DB2 Universal Database
IBM and the IBM logo

PS/2
Wake on LAN
WebSphere

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Magellan is a registered trademark of Datalogic Scanning, Inc.

SYMBOL a registered trademark of Symbol Technologies, Inc.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Celeron and Intel are trademarks of Intel corporation in the United States, or other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, or other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

Action classes 28
AEF API 9
AEF error handling 34
AEF infrastructure 107
AEFBase 14
AEFErrorHandler 197
AEFException 15
AEFSession 11
AEFSessionFactory 12
AEFSessionPool 13
Appautomation.properties 203
Appclass.properties 203
Appconfig.properties 204
application descriptors file 201
application events, POS 182
application hooks, POS 182
application, POS, events, XML 235
architecture, understanding the 11
attributes 69
Authorization Properties (Common) 212
Authorization Properties (GSA) 213
Authorization Properties (SA and ACE) 214
automatic manager override number 181
Automation API 28
Automation bundle 203
  Appautomation.properties 203
  Automation.properties 203
Automation Bundle 199
Automation Provider 19, 125
Automation Provider API 186
Automation.properties 203

B

backup controller, states of a primary and 86
Bad conditions 33
battery return program 297
build, debug, package 175
business component architecture 5

C

cable ferrite requirement 296
cashReceipt 235
CashReceipt event 235
CashReceiptImage objects 56
Checkpoint file 59
Class A compliance statement
  Australia and New Zealand 294
  China 294
  European Union 292, 293
  FCC (USA) 292
  Germany 293
  Industry Canada 292
  Japan 295
  Russia 294
Class Bundle 200, 203
  Appclass.properties 203
  Classes.properties 203
Classes.properties 203
client configuration 61
client exception handling 60
client scenarios 58
code, SA user exit 193
coding guidelines 157
Component configurations 15
ConditionLock 34
conditions, error 197
Config.properties 204
configuration 61
Configuration Bundle 199, 204
  Appconfig.properties 204
  Config.properties 204
configuration properties 17
configuration properties, automation provider 24
configuration settings 181
configurations, component 15
controller
  backup
    full failover 81
    partial failover 81
  messages 84
  ownership matrix 85
  remote
    fails during normal operation 82
  switch back notification 84
controller, states of a primary and backup 86
coupon 236
Coupon event 236
Coupon interface 27
CouponProperties 207
customer 249
Customer interface 26
CustomerProperties 209

D

damage from electrostatic discharge 296
Data Provider 40, 111
Data Provider properties 207
dataEvent 239
DataEvent 239
debug, build, package 175
debugger, 4690 178
default passwords 181
determining the error key 35
Discount events 240
Discount interface 27
DiscountProperties 209
disposal of equipment 296

E

editing property files 203
electromagnetic Interference statement
  Russia 294
electronic emissions notices 292
  Australia and New Zealand 294
  China 294
  European Union 292, 293
  FCC (USA) 292
  Germany 293
  Industry Canada 292
  Japan 295
  Korea 295
electrostatic discharge (ESD) 296
elements 70
enabling a POS application 181
end of life disposal 296
environment setup 93
environment setup, POSBC 189
equipment disposal 296
Error Bundle 200, 204
error handling 34
Error handling 197
  Automation Bundle 199
  Class Bundle 200
  Configuration Bundle 199
  Error Bundle 200
  Function Code Bundle 200
  Good and bad conditions 197
  Key Sequence Bundle 201
  State Bundle 201
  Substate Bundle 201
Error handling modes, Automation Provider 20
error handling settings, examples of 22
error helper 159
Error helper classes 36
error helpers 202
error key 35
error logging 157
European Union battery recycling statement 298
event and listener types 43
events, XML 235
example of action classes 273
examples of error handling settings 22
exception handling 167
exception log, POSServer 59
exceptions and errors 60
exit code, SA user 193
Extending POSAppEvents 47

F

failover, partial 81
failover, understanding 80
ferrite requirement 296
files, persistence 81
flat panel displays 299
full failover to a backup controller 81
Function Code Bundle 200, 204

G

guidelines for coding 157

H

handling exceptions 167
helper, error 159
hierarchy, user 94
hierarchy, user, POSProvider/POSServer 189
hooks, application, POS 182

I

identifier arguments 22
info objects 23
infrastructure 11
infrastructure, AEF 107
infrastructure, POSBC 189
interaction diagrams 71
interactions 81
Internationalization Bundle 204
Item interface 27
itemEvent 242
ItemSales events 242
ItemSalesProperties 210

J

Japan Electronics and Information Technology Industries Association statement 295
Japanese Electrical Appliance and Material Safety Law statement 294
Japanese power line harmonics compliance statement 295
Japanese VCCI Council Class A statement 295
Java Development Kit 93
JDK 93

K

Key Sequence Bundle 201, 204
key sequence, modifying 135
KioskPOSServer 55
Korean communications statement 295

L

LineItem interface 27
lineItemDiscount 241
LineItemProperties 212
listener considerations 48
listener types 43
LoadBalancer 14
lock 249
log, exception, POSServer 59
Logon Bundle 205
logs, error 157

M

manager override number 181
mercury-added statement 299
message, new SI XML 114

N

new SI XML message 114
notices 291
  battery recycling 297, 298
  cable ferrites 296
  electronic emissions 292
  electrostatic discharge (ESD) 296
  end of life disposal 296
  Toshiba 291

O

ObjectDetectorLock 34
Operator events 246
Operator interface 26
OperatorAuthorizationProperties 212
Options 251
Options event 251
OptionsProperties 215
other documents xiv
overview, AEF architecture 2
overview, posprovider/POSServer api 5
overview, product 1
ownership matrix 85

P

package, build, debug 175
packaging your code 177
partial failover 81
partial failover to a backup controller 81
passwords, default 181
Payment integrity 57
perchlorate 299
persistence files 81
points 259
Points event 259
Points interface 27
PointsProperties 218
POS application XML events 235
POS application, enabling 181
POS device support 287
POS Request objects 56
POS Sales Application 9
POSAppEvents, extending 47
POSAutomationProvider 26
POSBC configuration 79
POSBC infrastructure 189
POSDeviceProperties 218
POSInfo objects 55
POSProvider 9, 54
POSProvider configuration 61
POSProvider/POSServer API 51
POSprovider/POSServer api overview
  server api overview, posprovider/pos 5
POSProvider/POSServer sample file 189
POSProvider/POSServer user hierarchy 189
POSProviderFactory 55
POSRequestFactory 55
POSResult objects 57
POSServer Exception Log 59
POSServerFactory 55
PriceInfo objects 56
primary and backup controller, states of a 86
PrintLine objects 56
procedure definitions 58
properties file configuration 175
properties, configuration 17
properties, data provider 207
  Authorization Properties (Common) 212
  Authorization Properties (GSA) 213
  Authorization Properties (SA and ACE) 214
  CouponProperties 207
  CustomerProperties 209
  DiscountProperties 209
  ItemSalesProperties 210
  LineItemProperties 212
  OperatorAuthorizationProperties 212, 213, 214
  OptionsProperties 215
  PointsProperties 218
  POSDeviceProperties 218
  StoreOptionsProperties 222
  TenderProperties 227
  TerminalOptionsProperties 227
  TransactionStatusProperties 229
  TransactionTotalsProperties 232
  WorkstationStatusProperties 233
properties, Data Provider 40
Property conditions 31
publications, related xiv

R

related publications xiv
remote controller fails during normal operation 82
remote interfaces 28
remote procedure call, XML 64
report 260
Report event 260
requirements for user exits 187
ResultLineItem objects 56
running the SampleGUI client 106

S

SA user exit code 193
safety information xi
sales application 9
SalesTransaction 55
SalesTransaction interface 26
SALogonActionImpl 273
sample POSProvider/POSServerfile 189
SampleGUI client 106
scale 260
Scale event 260
Session Bundle 205
sessions, real or virtual 2
SessionServer 13

settings, examples of error handling 22
SI XML message, new 114
signOff 248
signOn 246
soeps header 64
solution overview 80
State Bundle 201, 205
states of a primary and backup controller 86
StoreOptionsProperties 222
Substate Bundle 201, 205
support, POS device 287
switch back to primary controller 82

T

Taiwanese battery recycling statement 297
tasks, AEF programming 4
Telecommunications regulatory statement 292
Tender 260
Tender event 260
Tender interface 28
Tender Map Bundle 205
tender mappings 181
TenderProperties 227
TerminalOptionsProperties 227
third party data change 85
third party ownership voting 85
trademarks 299
Transaction interface 26
transactionDiscount 240
transactionEnd 267
TransactionReplayComplete event 271
transactionStart 263
TransactionStatus events 263
TransactionStatusProperties 229
transactionSuspended 266
transactionTaxChange 269
transactionTotals 270
TransactionTotals event 270
TransactionTotalsProperties 232
transactionUpdate 269
transactionVoid 265

U

unlock 249
user exit code, SA 193
user extensions 186
user hierarchy 94
using the 4690 debugger 178

V

virtual receipts 69

W

workstationStatus 271
WorkstationStatus event 271
WorkstationStatusProperties 233

X

XML events 235
  CashReceipt 235
  coupon 236
  dataEvent 239
  Discount events
    lineItemDiscount 240
    transactionDiscount 240
  itemEvent 242
  Operator events
    customer 246
    lock 246
    signOff 246
    signOn 246
    unLock 246
  Options 251
  Points 259
  report 260
  Scale 260
  Tender 260
  TransactionReplayComplete 271
  TransactionStatus events
    transactionEnd 263
    transactionStart 263
    transactionSuspended 263
    transactionTaxChange 263
    transactionUpdate 263
    transactionVoid 263
  transactionTotals 270
  workstationStatus 271
XML message, new SI 114
XML remote procedure call 64

TOSHIBA
Leading Innovation

Printed in USA

G362-0562-05